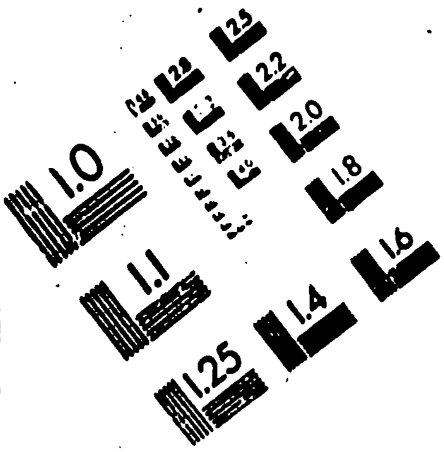


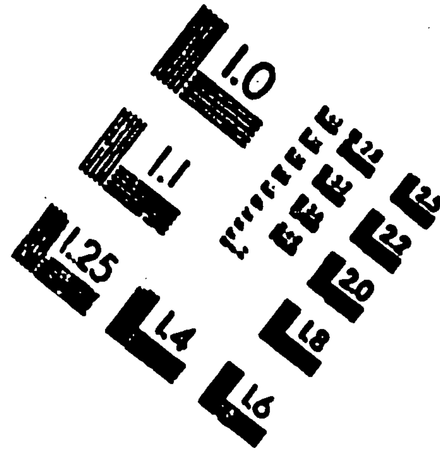
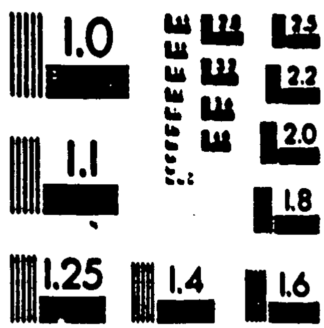
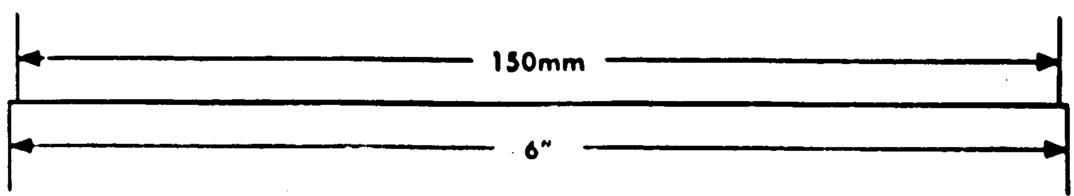
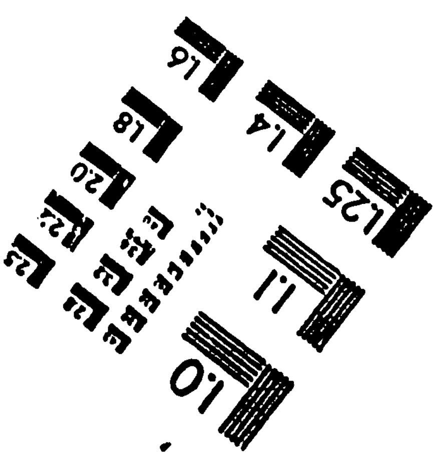
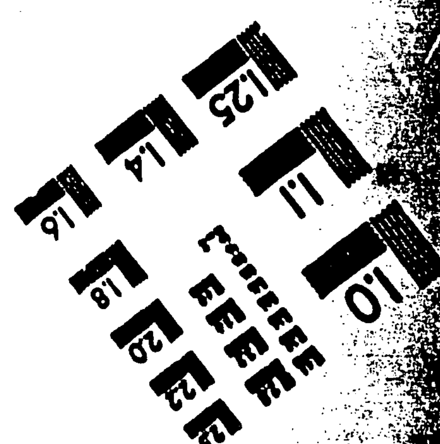


IMAGE EVALUATION TEST TARGET (MT-3)









PHOTOGRAPHIC SCIENCES CORPORATION
770 BASKET ROAD
P.O. BOX 338
WEBSTER, NEW YORK 14580
(716) 265-1600

microunity

Zeus System Architecture

MicroUnity

Craig Hansen
Cnief Architect

MicroUnity Systems Engineering, Inc.
475 Potrero Avenue
Sunnyvale, CA 94086.4118
Phone: 408.734.8100
Fax: 408.734.8136
email: craig@n.icrounity.com
http://www.microunity.com

Contents

Tables and Figures

3

Introduction

MicroUnity's Zeus Architecture describes general-purpose processor, memory, and interface subsystems, organized to operate at the enormously high bandwidth rates required for broadband applications.

The Zeus processor performs integer, floating point, signal processing and non-linear operations such as Galois field, table lookup and bit switching on data sizes from 1 bit to 128 bits. Group or SIMD (single instruction multiple data) operations sustain external operand bandwidth rates up to 512 bits (i.e., up to four 128-bit operand groups) per instruction even on data items of small size. The processor performs ensemble operations such as convolution that maintain full intermediate precision with aggregate internal operand bandwidth rates up to 20,000 bits per instruction. The processor performs wide operations such as crossbar switch, matrix multiply and table lookup that use caches embedded in the execution units themselves to extend operands to as much as 32768 bits. All instructions produce at most a single 128-bit register result, source at most three 128-bit registers and are free of side effects such as the setting of condition codes and flags. The instruction set design carries the concept of streamlining beyond Reduced Instruction Set Computer (RISC) architectures, to simplify implementations that issue several instructions per machine cycle.

The Zeus memory subsystem provides 64-bit virtual and physical addressing for UNIX, Mach, and other advanced OS environments. Separate address instructions enable the division of the processor into decoupled access and execution units, to reduce the effective latency of memory to the pipeline. The Zeus cache supplies the high data and instruction issue rates of the processor, and supports coherency primitives for scaleable multiprocessors. The memory subsystem includes mechanisms for sustaining high data rates not only in block transfer modes, but also in non-unit stride and scatterred access patterns.

The Zeus interface subsystem is designed to match industry standard "Socket 7" protocols and pin-outs. In this way, Zeus can make use of the immense infrastructure of the PC for building low-cost systems. The interface subsystem is modular, and can be replaced with appropriate protocols and pin-outs for lower-cost and higher-performance systems.

The goal of the Zeus architecture is to integrate these processor, memory, and interface capabilities with optimal simplicity and generality. From the software perspective, the entire machine state consists of a program counter, a single bank of 64 general-purpose 128-bit registers, and a linear byte-addressed shared memory space with mapped interface registers. All interrupts and exceptions are precise, and occur with low overhead.

This document is intended for Zeus software and hardware developers alike, and defines the interface at which their designs must meet. Zeus pursues the most efficient tradeoffs between hardware and software complexity by making all processor, memory, and interface resources directly accessible to high-level language programs.

Conformance

To ensure that Zeus systems may freely interchange data, user-level programs, system-level programs and interface devices, the Zeus system architecture reaches above the processor level architecture.

Mandatory and Optional Areas

A computer system conforms to the requirements of the Zeus System Architecture if and only if it implements all the specifications described in this document and other specifications included by reference. Conformance to the specification is mandatory in all areas, including the instruction set, memory management system, interface devices and external interfaces, and bootstrap ROM functional requirements, except where explicit options are stated.

Optional areas include:

Number of processor threads
Size of first-level cache memories
Existence of a second-level cache
Size of second-level cache memory
Size of system-level memory
Existence of certain optional interface device interfaces

Upward-compatible Modifications

From time to time, MicroUnity may modify the architecture in an upward-compatible manner, such as by the addition of new instructions, definition of reserved bits in system state, or addition of new standard interfaces. Such modifications will be added as options, so that designs that conform to this version of the architecture will conform to future, modified versions.

Additional devices and interfaces, not covered by this standard may be added in specified regions of the physical memory space, provided that system reset places these devices and interfaces in an inactive state that does not interfere with the operation of software that runs in any conformant system. The software interface requirements of any such additional devices and interfaces must be made as widely available as this architecture specification.

Promotion of Optional Features

It is most strongly recommended that such optional instructions, state or interfaces be implemented in all conforming designs. Such implementations enhance the value of the features in particular and the architecture as a whole by broadening the set of implementations over which software may depend upon the presence of these features.

Implementations that fail to implement these features may encounter unacceptable levels of overhead when attempting to emulate the features by exception handlers or use of virtual

memory. This is a particular concern when involved in code that has real-time performance constraints.

In order that upward-compatible optional extensions of the original Zeus system architecture may be relied upon by system and application software, MicroUnity may upon occasion promote optional features to mandatory conformance for implementations designed or produced after a suitable delay upon such notification by publication of future version of the specification.

Unrestricted Physical Implementation

Nothing in this specification should be construed to limit the implementation choices of the conforming system beyond the specific requirements stated herein. In particular, a computer system may conform to the Zeus System Architecture while employing any number of components, dissipate any amount of heat, require any special environmental facilities, or be of any physical size.

Draft Version

This document is a draft version of the architectural specification. In this form, conformance to this document may not be claimed or implied. MicroUnity may change this specification at any time, in any manner, until it has been declared final. When this document has been declared final, the only changes will be to correct bugs, defects or deficiencies, and to add upward-compatible optional extensions.

Common Elements

Notation

The descriptive notation used in this document is summarized in the table below:

$x + y$	two's complement addition of x and y. Result is the same size as the operands, and operands must be of equal size.
$x - y$	two's complement subtraction of y from x. Result is the same size as the operands, and operands must be of equal size.
$x * y$	two's complement multiplication of x and y. Result is the same size as the operands, and operands must be of equal size.
x / y	two's complement division of x by y. Result is the same size as the operands, and operands must be of equal size.
$x \& y$	bitwise and of x and y. Result is same size as the operands, and operands must be of equal size.
$x \mid y$	bitwise or of x and y. Result is same size as the operands, and operands must be of equal size.
$x \hat{} y$	bitwise exclusive-of of x and y. Result is same size as the operands, and operands must be of equal size.
$\sim x$	bitwise inversion of x. Result is same size as the operand.
$x = y$	two's complement equality comparison between x and y. Result is a single bit, and operands must be of equal size.
$x \neq y$	two's complement inequality comparison between x and y. Result is a single bit, and operands must be of equal size.
$x < y$	two's complement less than comparison between x and y. Result is a single bit, and operands must be of equal size.
$x \geq y$	two's complement greater than or equal comparison between x and y. Result is a single bit, and operands must be of equal size.
$\sqrt{x}$	floating-point square root of x
$x \| y$	concatenation of bit field x to left of bit field y
x^y	binary digit x repeated, concatenated y times. Size of result is y.
x_y	extraction of bit y (using little-endian bit numbering) from value x. Result is a single bit.
$x_{y..z}$	extraction of bit field formed from bits y through z of value x. Size of result is y-z+1; if z>y, result is an empty string.
$x?y:z$	value of y, if x is true, otherwise value of z. Value of x is a single bit.
$x \leftarrow y$	bitwise assignment of x to value of y
Sn	signed, two's complement, binary data format of n bytes
Un	unsigned binary data format of n bytes
Fn	floating-point data format of n bytes

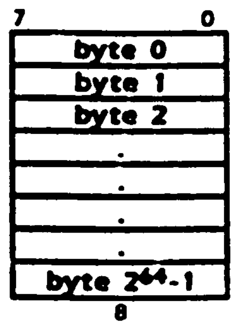
descriptive notation

Bit ordering

The ordering of bits in this document is always little-endian, regardless of the ordering of bytes within larger data structures. Thus, the least-significant bit of a data structure is always labeled 0 (zero), and the most-significant bit is labeled as the data structure size (in bits) minus one.

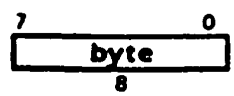
Memory

Zeus memory is an array of 2^{64} bytes, without a specified byte ordering, which is physically distributed among various components.



Byte

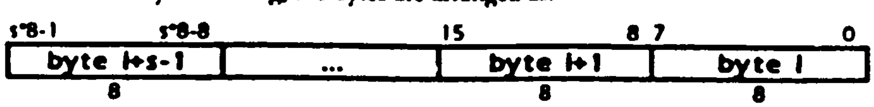
A byte is a single element of the memory array, consisting of 8 bits:



Byte ordering

Larger data structures are constructed from the concatenation of bytes in either little-endian or big-endian byte ordering. A memory access of a data structure of size s at address i is formed from memory bytes at addresses i through i+s-1. Unless otherwise specified, there is no specific requirement of alignment: it is not generally required that i be a multiple of s. Aligned accesses are preferred whenever possible, however, as they will often require one fewer processor or memory clock cycle than unaligned accesses.

With little-endian byte ordering, the bytes are arranged as:



With big-endian byte ordering, the bytes are arranged as:

s*8-1 s*8-8	s*8-9 s*8-16		7 0
byte i	byte i+1	...	byte i+s-1
8	8		8

Zeus memory is byte-addressed, using either little-endian or big-endian byte ordering. For consistency with the bit ordering, and for compatibility with x86 processors, Zeus uses little-endian byte ordering when an ordering must be selected. Zeus load and store instructions are available for both little-endian and big-endian byte ordering. The selection of byte ordering is dynamic, so that little-endian and big-endian processes, and even data structures within a process, can be intermixed on the processor.

Memory read/load semantics

Zeus memory, including memory-mapped registers, must conform to the following requirements regarding side-effects of read or load operations:

A memory read must have no side-effects on the contents of the addressed memory nor on the contents of any other memory.

Memory write/store semantics

Zeus memory, including memory-mapped registers, must conform to the following requirements regarding side-effects of read or load operations:

A memory write must affect the contents of the addressed memory so that a memory read of the addressed memory returns the value written, and so that a memory read of a portion of the addressed memory returns the appropriate portion of the value written.

A memory write may affect or cause side-effects on the contents of memory not addressed by the write operation, however, a second memory write of the same value to the same address must have no side-effects on any memory; memory write operations must be idempotent.

Zeus store instructions that are weakly ordered may have side-effects on the contents of memory not addressed by the store itself; subsequent load instructions which are also weakly ordered may or may not return values which reflect the side-effects.

Data

Zeus provides eight-byte (64-bit) virtual and physical address sizes, and eight-byte (64-bit) and sixteen-byte (128-bit) data path sizes, and uses fixed-length four-byte (32-bit) instructions. Arithmetic is performed on two's-complement or unsigned binary and ANSI/IEEE standard 754-1985 conforming binary floating-point number representations.

6

Fixed-point Data

Bit

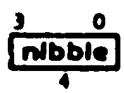
A bit is a primitive data element:

Peck

A peck is the catenation of two bits:

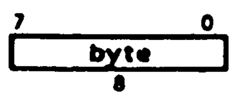
Nibble

A nibble is the catenation of four bits:



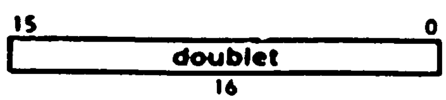
Byte

A byte is the catenation of eight bits, and is a single element of the memory array:



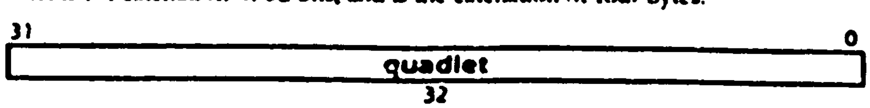
Doublet

A doublet is the catenation of 16 bits, and is the catenation of two bytes:



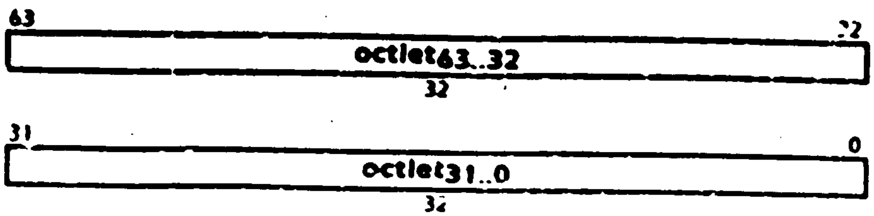
Quadlet

A quadlet is the catenation of 32 bits, and is the catenation of four bytes:



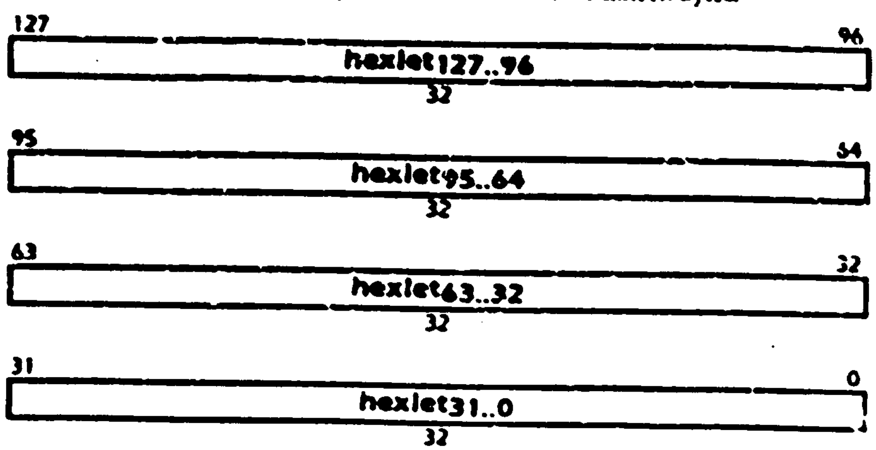
Octlet

An octlet is the catenation of 64 bits, and is the catenation of eight bytes:



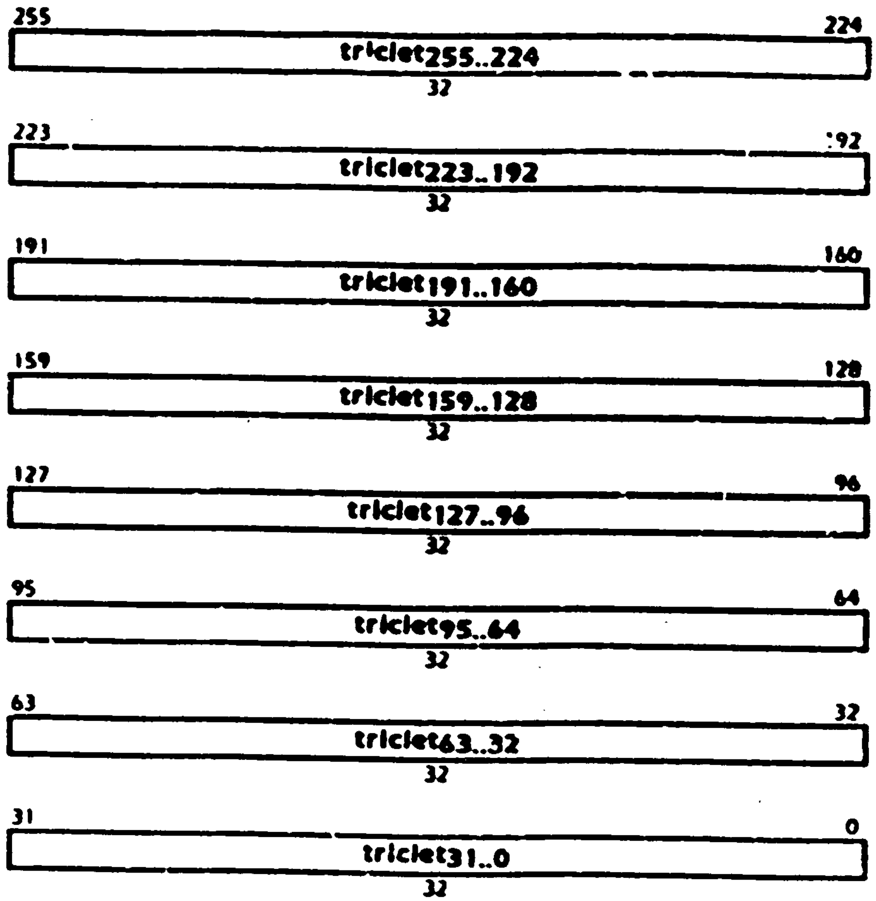
Hexlet

A hexlet is the catenation of 128 bits, and is the catenation of sixteen bytes:



Tricklet

A tricklet is the catenation of 256 bits, and is the catenation of thirty-two bytes:



Address

Zeus addresses, both virtual addresses and physical addresses, are octlet quantities.

Floating-point Data

Zeus's floating-point formats are designed to satisfy ANSI/IEEE standard 754-1985: Binary Floating-point Arithmetic. Standard 754 leaves certain aspects to the discretion of implementers: additional precision formats, encoding of quiet and signaling NaN values, details of production and propagation of quiet NaN values. These aspects are detailed below.

Zeus adds additional half-precision and quad-precision formats to standard 754's single-precision and double-precision formats. Zeus's double-precision satisfies standard 754's

precision requirements for a single extended format, and Zeus's quad precision satisfies standard 754's precision requirements for a double extended format.

Each precision format employs fields labeled s (sign), e (exponent), and f (fraction) to encode values that are (1) NaN: quiet and signaling, (2) infinities: $(-1)^{s}\infty$, (3) normalized numbers: $(-1)^{s}2^{e-bias}(1.f)$, (4) denormalized numbers: $(-1)^{s}2^{1-bias}(0.f)$, and (5) zero: $(-1)^{s}0$.

Quiet NaN values are denoted by any sign bit value, an exponent field of all one bits, and a non-zero fraction with the most significant bit set. Quiet NaN values generated by default exception handling of standard operations have a zero sign bit, an exponent field of all one bits, a fraction field with the most significant bit set, and all other bits cleared.

Signaling NaN values are denoted by any sign bit value, an exponent field of all one bits, and a non-zero fraction with the most significant bit cleared.

Infinite values are denoted by any sign bit value, an exponent field of all one bits, and a zero fraction field.

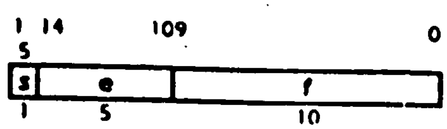
Normalized number values are denoted by any sign bit value, an exponent field that is not all one bits or all zero bits, and any fraction field value. The numeric value encoded is $(-1)^{s}2^{e-bias}(1.f)$. The bias is equal the value resulting from setting all but the most significant bit of the exponent field, half: 15, single: 127, double: 1023, and quad: 16383.

Denormalized number values are denoted by any sign bit value, an exponent field that is all zero bits, and a non-zero fraction field value. The numeric value encoded is $(-1)^{s}2^{1-bias}(0.f)$.

Zero values are denoted by any sign bit value, and exponent field that is all zero bits, and a fraction field that is all zero bits. The numeric value encoded is $(-1)^{s}0$. The distinction between +0 and -0 is significant in some operations.

Half-precision Floating-point

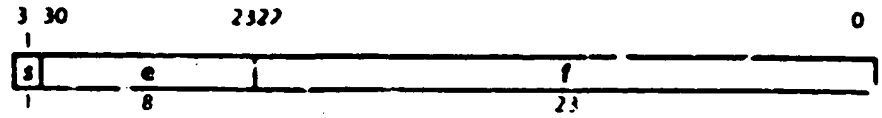
Zeus half precision uses a format similar to standard 754's requirements, reduced to a 16 bit overall format. The format contains sufficient precision and exponent range to hold a 12 bit signed integer.



9

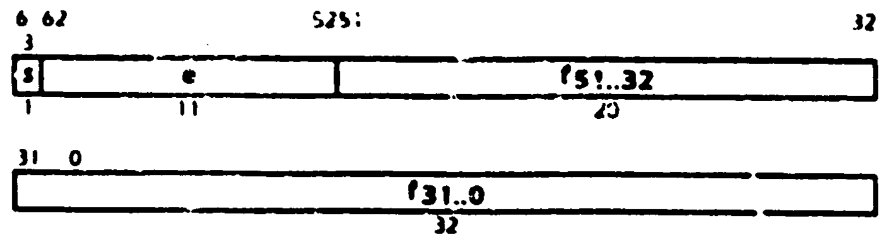
Single-precision Floating-point

Zeus single precision satisfies standard 754's requirements for "single."



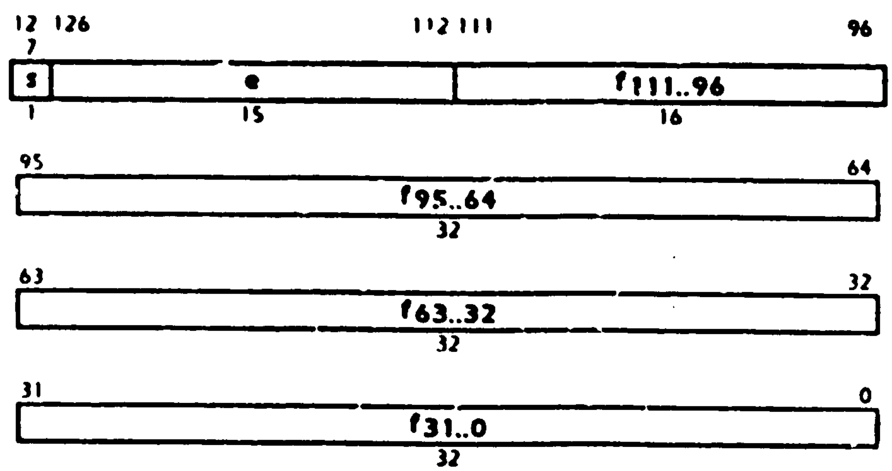
Double-precision Floating-point

Zeus double precision satisfies standard 754's requirements for "double."



Quad-precision Floating-point

Zeus quad precision satisfies standard 754's requirements for "double extended" but has additional fraction precision to use 128 bits.



Zeus Processor

MicroUnity's Zeus processor provides the general-purpose, high-bandwidth computation capability of the Zeus system. Zeus includes high-bandwidth data paths, register files, and a memory hierarchy. Zeus's memory hierarchy includes on-chip instruction and data memories, instruction and data caches, a virtual memory facility, and interfaces to external devices. Zeus's interfaces in the initial implementation are solely the "Super Socket 7" bus, but other implementations may have different or additional interfaces.

Architectural Framework

The Zeus architecture defines a compatible framework for a family of implementations with a range of capabilities. The following implementation-defined parameters are used in the rest of the document in boldface. The value indicated is for MicroUnity's first Zeus implementation.

Parameter	Interpretation	Value	Range of legal values
T	number of execution threads	4	$1 \le T \le 31$
CE	$\log_2$ cache blocks in first-level cache	9	$0 \le CE \le 31$
CS	$\log_2$ cache blocks in first-level cache set	2	$0 \le CS \le 4$
CT	existence of dedicated tags in first-level cache	1	$0 \le CT \le 1$
LE	$\log_2$ entries in local TB	0	$0 \le LE \le 3$
LB	Local TB based on base register	1	$0 \le LB \le 1$
GE	$\log_2$ entries in global TB	7	$0 \le GE \le 15$
GT	$\log_2$ threads which share a global TB	1	$0 \le GT \le 3$

Interfaces and Block Diagram

The first implementation of Zeus uses "socket 7" protocols and pinouts.

Instruction

Assembler Syntax

Instructions are specified to Zeus assemblers and other code tools (assemblers) in the syntax of an instruction mnemonic (operation code), then optionally white space (blanks or tabs) followed by a list of operands.

The instruction mnemonics listed in this specification are in upper case (capital) letters, assemblers accept either upper case or lower case letters in the instruction mnemonics. In

9

this specification, instruction mnemonics contain periods (".") to separate elements to make them easier to understand; assemblers ignore periods within instruction mnemonics. The instruction mnemonics are designed to be parsed uniquely without the separating periods.

If the instruction produces a register result, this operand is listed first. Following this operand, if there are one or more source operands, is a separator which may be a comma (","), equal ("="), or at-sign ("@"). The equal separates the result operand from the source operands, and may optionally be expressed as a comma in assembler code. The at-sign indicates that the result operand is also a source operand, and may optionally be expressed as a comma in assembler code. If the instruction specification has an equal-sign, an at-sign in assembler code indicates that the result operand should be repeated as the first source operand (for example, "A.ADD.I r4@5" is equivalent to "A.ADD.I r4=r4,5"). Commas always separate the remaining source operands.

The result and source operands are case-sensitive; upper case and lower case letters are distinct. Register operands are specified by the names r0 (or r00) through r63 (a lower case "r" immediately followed by a one or two digit number from 0 to 63), or by the special designations of "lp" for "r0," "dp" for "r1," "fp" for "r62," and "sp" for "r63." Integer-valued operands are specified by an optional sign (-) or (+) followed by a number, and assemblers generally accept a variety of integer-valued expressions.

Instruction Structure

A Zeus instruction is specifically defined as a four-byte structure with the little-endian ordering shown below. It is different from the quadlet defined above because the placement of instructions into memory must be independent of the byte ordering used for data structures. Instructions must be aligned on four-byte boundaries; in the diagram below, i must be a multiple of 4.

31 … 24	23 … 16	15 … 8	7 … 0
byte i+3	byte i+2	byte i+1	byte i
8	8	8	8

Gateway

A Zeus gateway is specifically defined as an 8-byte structure with the little-endian ordering shown below. A gateway contains a code address used to securely invoke a system call or procedure at a higher privilege level. Gateways are marked by protection information specified in the TB. Gateways must be aligned on 8-byte boundaries; in the diagram below, i must be a multiple of 8.

63 … 56	55 … 48	47 … 40	39 … 32
byte i+7	byte i+6	byte i+5	byte i+4
8	8	8	8

31 … 24	23 … 16	15 … 8	7 … 0
byte i+3	byte i+2	byte i+1	byte i
8	8	8	8

The gateway contains two data items within its structure, a code address and a new privilege level:

63 … 2	1 0
code address	pl
62	2

The virtual memory system can be used to designate a region of memory as containing gateways. Other data may be placed within the gateway region, provided that if an attempt is made to use the additional data as a gateway, that security cannot be violated. For example, 64-bit data or stack pointers which are aligned to at least 4 bytes and are in little-endian byte order have pl=0, so that the privilege level cannot be raised by attempting to use the additional data as a gateway.

User State

The user state consists of hardware data structures that are accessible to all conventional compiled code. The Zeus user state is designed to be as regular as possible, and consists only of the general registers, the program counter, and virtual memory. There are no specialized registers for condition codes, operating modes, rounding modes, integer multiply/divide, or floating-point values.

General Registers

Zeus user state includes 64 general registers. All are identical; there is no dedicated zero-valued register, and there are no dedicated floating-point registers.

127 … 0
REG[0]
REG[1]
REG[2]
⋮
REG[62]
REG[63]
128

Some Zeus instructions have 64 bit register operands. These operands are sign-extended to 128 bits when written to the register file, and the low-order 64 bits are chosen when read from the register file.

Definition

```
def val ← RegRead(rn, size)
    case size of
        64:
            val ← REG[rn]63..0
        128:
```

10


```
            val ← REG[rn]
    endcase
enddef

def RegWrite(rn, size, val)
    case size of
        64:
            REG[rn] ← val63^64 || val63..0
        128:
            REG[rn] ← val127..0
    endcase
enddef
```

Program Counter

The program counter contains the address of the currently executing instruction. This register is implicitly manipulated by branch instructions, and read by branch instructions that save a return address in a general register.

63 … 2	1 0
ProgramCounter	0
62	2

Privilege Level

The privilege level register contains the privilege level of the currently executing instruction. This register is implicitly manipulated by branch gateway and branch down instructions, and read by branch gateway instructions that save a return address in a general register.

1 0
pl
2

Program Counter and Privilege Level

The program counter and privilege level may be packed into a single octlet. This combined data structure is saved by the Branch Gateway instruction and restored by the Branch Down instruction.

63 … 2	1 0
ProgramCounter	pl
62	2

System state

The system state consists of the facilities not normally used by conventional compiled code. These facilities provide mechanisms to execute such code in a fully virtual environment. All system state is memory mapped, so that it can be manipulated by compiled code.

Fixed-point

Zeus provides load and store instructions to move data between memory and the registers, branch instructions to compare the contents of registers and to transfer control from one code address to another, and arithmetic operations to perform computation on the contents of registers, returning the result to registers.

Load and Store

The load and store instructions move data between memory and the registers. When loading data from memory into a register, values are zero-extended or sign-extended to fill the register. When storing data from a register into memory, values are truncated on the left to fit the specified memory region.

Load and store instructions that specify a memory region of more than one byte may use either little-endian or big-endian byte ordering: the size and ordering are explicitly specified in the instruction. Regions larger than one byte may be either aligned to addresses that are an even multiple of the size of the region or of unspecified alignment: alignment checking is also explicitly specified in the instruction.

Load and store instructions specify memory addresses as the sum of a base general register and the product of the size of the memory region and either an immediate value or another general register. Scaling maximizes the memory space which can be reached by immediate offsets from a single base general register, and assists in generating memory addresses within iterative loops. Alignment of the address can be reduced to checking the alignment of the first general register.

The load and store instructions are used for fixed-point data as well as floating-point and digital signal processing data; Zeus has a single bank of registers for all data types.

Swap instructions provide multithread and multiprocessor synchronization, using indivisible operations: add-swap, compare-swap, multiplex-swap, and double-compare-swap. A store-multiplex operation provides the ability to indivisibly write to a portion of an octlet. These instructions always operate on aligned octlet data, using either little-endian or big-endian byte ordering.

Branch

The fixed-point compare-and-branch instructions provide all arithmetic tests for equality and inequality of signed and unsigned fixed-point values. Tests are performed either between two operands contained in general registers, or on the bitwise and of two operands. Depending on the result of the compare, either a branch is taken, or not taken. A taken branch causes an immediate transfer of the program counter to the target of the branch, specified by a 12-bit signed offset from the location of the branch instruction. A non-taken branch causes no transfer; execution continues with the following instruction.

Other branch instructions provide for unconditional transfer of control to addresses too distant to be reached by a 12-bit offset, and to transfer to a target while placing the location

following the branch into a register. The branch through gateway instruction provides a secure means to access code at a higher privilege level, in a form similar to a normal procedure call.

Addressing Operations

A subset of general fixed-point arithmetic operations is available as addressing operations. These include add, subtract, Boolean, and simple shift operations. These addressing operations may be performed at a point in the Zeus processor pipeline so that they may be completed prior to or in conjunction with the execution of load and store operations in a "superspring" pipeline in which other arithmetic operations are deferred until the completion of load and store operations.

Execution Operations

Many of the operations used for Digital Signal Processing (DSP), which are described in greater detail below, are also used for performing simple scalar operations. These operations perform arithmetic operations on values of 8-, 16-, 32-, 64-, or 128- bit sizes, which are right-aligned in registers. These execution operations include the add, subtract, boolean and simple shift operations which are also available as addressing operations, but further extend the available set to include three-operand add/subtract, three-operand boolean, dynamic shifts, and bit-field operations.

Floating-point

Zeus provides all the facilities mandated and recommended by ANSI/IEEE standard 754-1985: Binary Floating-point Arithmetic, with the use of supporting software.

Branch Conditionally

The floating-point compare-and-branch instructions provide all the comparison types required and suggested by the IEEE floating-point standard. These floating-point comparisons augment the usual types of numeric value comparisons with special handling for NaN (not-a-number) values. A NaN value compares as "unordered" with respect to any other value, even that of an identical NaN value.

Zeus floating-point compare-branch instructions do not generate an exception on comparisons involving quiet or signaling NaN values. If such exceptions are desired, they can be obtained by combining the use of a floating-point compare-set instruction, with either a floating-point compare-branch instruction on the floating-point operands or a fixed-point compare-branch on the set result.

Because the less and greater relations are anti-commutative, one of each relation that differs from another only by the replacement of an L with a G in the code can be removed by reversing the order of the operands and using the other code. Thus, an L relation can be used in place of a G relation by swapping the operands to the compare-branch or compare-set instruction.

No instructions are provided that branch when the values are unordered. To accomplish such an operation, use the reverse condition to branch over an immediately following unconditional branch, or in the case of an if-then-else clause, reverse the clauses and use the reverse condition.

The E relation can be used to determine the unordered condition of a single operand by comparing the operand with itself.

The following floating-point compare-branch relations are provided as instructions:

Mnemonic		Branch taken if values compare as:				Exception if	
code	C-like	Unord-ered	Greater	Less	Equal	unord-ered	invalid
E	==	F	F	F	T	no	no
LG	<>	F	T	T	F	no	no
L	<	F	F	T	F	no	no
GE	>=	F	T	F	T	no	no

compare-branch relations

Compare-set

The compare-set floating-point instructions provide all the comparison types supported as branch instructions. Zeus compare-set floating-point instructions may optionally generate an exception on comparisons involving quiet or signaling NaNs.

The following floating-point compare-set relations are provided as instructions:

Mnemonic		Result if values compare as:				Exception if	
code	C-like	Unord-ered	Greater	Less	Equal	unord-ered	invalid
E	==	F	F	F	T	no	no
LG	<>	F	T	T	F	no	no
L	<	F	F	T	F	no	no
GE	>=	F	T	F	T	no	no
E.X	==	F	F	F	T	no	yes
LG.X	<>	F	T	T	F	no	yes
L.X	<	F	F	T	F	yes	yes
GE.X	<=	F	T	F	T	yes	yes

compare-set relations

Arithmetic Operations

The basic operations supported in hardware are floating-point add, subtract, multiply, divide, square root and conversions among floating-point formats and between floating-point and binary integer formats.

Software libraries provide other operations required by the ANSI/IEEE floating-point standard.

The operations explicitly specify the precision of the operation, and round the result (or check that the result is exact) to the specified precision at the conclusion of each operation. Each of the basic operations splits operand registers into symbols of the specified precision and performs the same operation on corresponding symbols.

In addition to the basic operations, Zeus performs a variety of operations in which one or more products are summed to each other and/or to an additional operand. The instructions include a fused multiply-add (E.MUL.ADD.F), convolve (E.CON.F), matrix multiply (E.MUL.MAT.F), and scale-add (E.SCAL.ADD.F).

The results of these operations are computed as if the multiplies are performed to infinite precision, added as if in infinite precision, then rounded only once. Consequently, these operations perform these operations with no rounding of intermediate results that would have limited the accuracy of the result

Rounding and exceptions

Rounding is specified within the instructions explicitly, to avoid explicit state registers for a rounding mode. Similarly, the instructions explicitly specify how standard exceptions (invalid operation, division by zero, overflow, underflow and inexact) are to be handled.[1]

When no rounding is explicitly named by the instruction (default), round to nearest rounding is performed, and all floating-point exception signals cause the standard-specified default result, rather than a trap. When rounding is explicity named by the instruction (N: nearest, Z: zero, F: floor, C: ceiling), the specified rounding is performed, and floating-point exception signals other than inexact cause a floating-point exception trap. When X (exact, or exception) is specified, all floating-point exception signals cause a floating-point exception trap, including inexact.

This technique assists the Zeus processor in executing floating-point operations with greater parallelism. When default rounding and exception handling control is specified in floating-point instructions, Zeus may safely retire instructions following them, as they are guaranteed not to cause data-dependent exceptions. Similarly, floating-point instructions with N, Z, F, or C control can be guaranteed not to cause data-dependent exceptions once the operands have been examined to rule out invalid operations, division by zero, overflow or underflow exceptions. Only floating-point instructions with X control or when exceptions cannot be ruled out with N, Z, F, or C control need to avoid retiring following instructions until the final result is generated.

ANSI/IEEE standard 754-1985 specifies information to be given to trap handlers for the five floating-point exceptions. The Zeus architecture produces a precise exception, (The program counter points to the instruction that caused the exception and all register state is present) from which all the required information can be produced in software, as all source operand values and the specfied operation are available.

---

[1] U.S. Patent 5,812,439 describes this "Technique of incorporating floating point information into processor instructions."

ANSI/IEEE standard 754-1985 specifies a set of five "sticky exception" bits, for recording the occurrence of exceptions that are handled by default. The Zeus architecture produces a precise exception for instructions with N, Z, F, or C control for invalid operation, division by zero, overflow or underflow exceptions and with X control for all floating-point exceptions, from which corresponding sticky-exception bits can be set. Execution of the same instruction with default control will compute the default result with round-to-nearest rounding. Most compound operations not specified by the standard are not available with rounding and exception controls.

NaN handling

ANSI/IEEE standard 754-1985 specifies that operations involving a signaling NaN or invalid operation shall, if no trap occurs and if a floating-point result is to be delivered, deliver a quiet NaN as its result. However, it fails to specify what quiet NaN value to deliver.

Zeus operations that produce a floating-point result and do not trap on invalid operations propagate signaling NaN values from operands to results, changing the signaling NaN values to quiet NaN values by setting the most significant fraction bit and leaving the remaining bits unchanged. Other causes of invalid operations produce the default quiet NaN value, where the sign bit is zero, the exponent field is all one bits, the most significant fraction bit is set and the remaing fraction bits are zero bits. For Zeus operations that produce multiple results catenated together, signaling NaN propagation or quiet NaN production is handled separately and independently for each result symbol.

ANSI/IEEE standard 754-1985 specifies that quiet NaN values should be propagated from operand to result by the basic operations. However, it fails to specify which of several quiet NaN values to propagate when more than one operand is a quiet NaN. In addition, the standard does not clearly specify how quiet NaN should be propagated for the multiple-operation instructions provided in Zeus. The standard does not specify the quiet NaN produced as a result of an operand being a signaling NaN when invalid operation exceptions are handled by default. The standard leaves unspecified how quiet and signaling NaN values are propagated though format conversions and the absolute-value, negate and copy operations. This section specifies these aspects left unspecified by the standard.

First of all, for Zeus operations that produce multiple results catenated together, quiet and signaling NaN propagation is handled separately and independently for each result symbol. A quiet or signaling NaN value in a single symbol of an operand causes only those result symbols that are dependent on that operand symbol's value to be propagated as that quiet NaN. Multiple quiet or signaling NaN values in symbols of an operand which influence separate symbols of the result are propagated independently of each other. Any signaling NaN that is propagated has the high-order fraction bit set to convert it to a quiet NaN.

For Zeus operations in which multiple symbols among operands upon which a result symbol is dependent are quiet or signaling NaNs, a priority rule will determine which NaN is propagated. Priority shall be given to the operand that is specified by a register definition at a lower-numbered (little-endian) bit position within the instruction (rb has priority over rc, which has priority over rd). In the case of operands which are catenated from two registers, priority shall be assigned based on the register which has highest priority (lower-numbered

bit position within the instruction). In the case of tie (as when the E.SCALADD scaling operand has two corresponding NaN values, or when a E.MUL.CF operand has NaN values for both real and imaginary components of a value), the value which is located at a lower-numbered (little-endian) bit position within the operand is to receive priority. The identification of a NaN as quiet or signaling shall not confer any priority for selection – only the operand position, though a signaling NaN will cause an invalid operand exception.

The sign bit of NaN values propagated shall be complemented if the instruction subtracts or negates the corresponding operand or (but not and) multiplies it by or divides it by or divides it into an operand which has the sign bit set, even if that operand is another NaN. If a NaN is both subtracted and multiplied by a negative value, the sign bit shall be propagated unchanged.

For Zeus operations that convert between two floating-point formats (INFLATE and DEFLATE), NaN values are propagated by preserving the sign and the most-significant fraction bits, except that the most-significant bit of a signalling NaN is set and (for DEFLATE) the least-significant fraction bit preserved is combined, via a logical-or of all fraction bits not preserved. All additional fraction bits (for INFLATE) are set to zero.

For Zeus operations that convert from a floating-point format to a fixed-point format (SINK), NaN values produce zero values (maximum-likelihood estimate). Infinity values produce the largest representable positive or negative fixed-point value that fits in the destination field. When exception traps are enabled, NaN or Infinity values produce a floating-point exception. Underflows do not occur in the SINK operation, they produce –1, 0 or +1, depending on rounding controls.

For absolute-value, negate, or copy operations, NaN values are propagated with the sign bit cleared, complemented, or copied, respectively. Signalling NaN values cause the Invalid operation exception, propagating a quieted NaN in corresponding symbol locations (default) or an exception, as specified by the instruction.

Floating-point functions

The following functions are defined for use within the detailed instruction definitions in the following section. In these functions an internal format represents infinite-precision floating-point values as a four-element structure consisting of (1) s (sign bit): 0 for positive, 1 for negative, (2) t (type): NORM, ZERO, SNAN, QNAN, INFINITY, (3) e (exponent), and (4) f: (fraction). The mathematical interpretation of a normal value places the binary point at the units of the fraction, adjusted by the exponent: (-1)^s*(2^e)*f. The function F converts a packed IEEE floating-point value into internal format. The function PackF converts an internal format back into IEEE floating-point format, with rounding and exception control.

Definition

```
def eb ← ebits(prec) as
    case pref of
        16:
            eb ← 5
        32:
```

```
            eb ← 8
        64:
            eb ← 11
        128:
            eb ← 15
    endcase
enddef

def eb ← ebias(prec) as
    eb ← 0 || 1^(ebits(prec)-1)
enddef

def fb ← fbits(prec) as
    fb ← prec - 1 - eb
enddef

def a ← F(prec, ai) as
    a.s ← ai_(prec-1)
    ae ← ai_(prec-2..fbits(prec))
    af ← ai_(fbits(prec)-1..0)
    if ae = 1^(ebits(prec)) then
        if af = 0 then
            a.t ← INFINITY
        elseif af_(fbits(prec)-1) then
            a.t ← SNaN
            a.e ← -fbits(prec)
            a.f ← 1 || af_(fbits(prec)-2..0)
        else
            a.t ← QNaN
            a.e ← -fbits(prec)
            a.f ← af
        endif
    elseif ae = 0 then
        if af = 0 then
            a.t ← ZERO
        else
            a.t ← NORM
            a.e ← 1-ebias(prec)-fbits(prec)
            a.f ← 0 || af
        endif
    else
        a.t ← NORM
        a.e ← ae-ebias(prec)-fbits(prec)
        a.f ← 1 || af
    endif
enddef

def a ← DEFAULTQNAN as
    a.s ← 0
    a.t ← QNAN
    a.e ← -1
    a.f ← 1
enddef
```

```
def a ← DEFAULTSNAN as
    a.s ← 0
    a.t ← SNAN
    a.e ← -1
    a.f ← 1
enddef

def fadd(a,b) as faddr(a,b,N) enddef

def c ← faddr(a,b,round) as
    if a.t=NORM and b.t=NORM then
        // d,e are a,b with exponent aligned and fraction adjusted
        if a.e > b.e then
            d ← a
            e.t ← b.t
            e.s ← b.s
            e.e ← a.e
            e.f ← b.f || 0^(a.e-b.e)
        else if a.e < b.e then
            d.t ← a.t
            d.s ← a.s
            d.e ← b.e
            d.f ← a.f || 0^(b.e-a.e)
            e ← b
        endif
        c.t ← d.t
        c.e ← d.e
        if d.s = e.s then
            c.s ← d.s
            c.f ← d.f + e.f
        elseif d.f > e.f then
            c.s ← d.s
            c.f ← d.f - e.f
        elseif d.f < e.f then
            c.s ← e.s
            c.f ← e.f - d.f
        else
            c.s ← r=F
            c.t ← ZERO
        endif
    // priority is given to b operand for NaN propagation
    elseif (b.t=SNAN) or (b.t=QNAN) then
        c ← b
    elseif (a.t=SNAN) or (a.t=QNAN) then
        c ← a
    elseif a.t=ZERO and b.t=ZERO then
        c.t ← ZERO
        c.s ← (a.s and b.s) or (round=F and (a.s or b.s))
    // NULL values are like zero, but do not combine with ZERO to alter sign
    elseif a.t=ZERO or a.t=NULL then
        c ← b
    elseif b.t=ZERO or b.t=NULL then
        c ← a
    elseif a.t=INFINITY and b.t=INFINITY then
```

```
            if a.s ≠ b.s then
                c ← DEFAULTSNAN // Invalid
            else
                c ← a
            endif
        elseif a.t=INFINITY then
            c ← a
        elseif b.t=INFINITY then
            c ← b
        else
            assert FALSE // should have covered al the cases above
        endif
enddef

def b ← fneg(a) as
        b.s ← ~a.s
        b.t ← a.t
        b.e ← a.e
        b.f ← a.f
enddef

def fsub(a,b) as fsubr(a,b,N) enddef

def fsubr(a,b,round) as faddr(a,fneg(b),round) enddef

def frsub(a,b) as frsubr(a,b,N) enddef

def frsubr(a,b,round) as faddr(fneg(a),b,round) enddef

def c ← fcom(a,b) as
        if (a.t=SNAN) or (a.t=QNAN) or (b.t=SNAN) or (b.t=QNAN) then
            c ← U
        elseif a.t=INFINITY and b.t=INFINITY then
            if a.s ≠ b.s then
                c ← (a.s=0) ? G: L
            else
                c ← E
            endif
        elseif a.t=INFINITY then
            c ← (a.s=0) ? G: L
        elseif b.t=INFINITY then
            c ← (b.s=0) ? G: L
        elseif a.t=NORM and b.t=NORM then
            if a.s ≠ b.s then
                c ← (a.s=0) ? G: L
            else
                if a.e > b.e then
                    af ← a.f
                    bf ← b.f || 0^(a.e-b.e)
                else
                    af ← a.f || 0^(b.e-a.e)
                    bf ← b.f
                endif
                if af = bf then
                    c ← E
```

```
            else
                c ← ((a.s=0) ^ (a.f > b.f)) ? G : L
            endif
        endif
    elseif a.t=NORM then
        c ← (a.s=0) ? G: L
    elseif b.t=NORM then
        c ← (b.s=0) ? G: L
    elseif a.t=ZERO and b.t=ZERO then
        c ← E
    else
        assert FALSE // should have covered al the cases above
    endif
enddef

def c ← fmul(a,b) as
    if a.t=NORM and b.t=NORM then
        c.s ← a.s ^ b.s
        c.t ← NORM
        c.e ← a.e + b.e
        c.f ← a.f * b.f
    // priority is given to b operand for NaN propagation
    elseif (b.t=SNAN) or (b.t=QNAN) then
        c.s ← a.s ^ b.s
        c.t ← b.t
        c.e ← b.e
        c.f ← b.f
    elseif (a.t=SNAN) or (a.t=QNAN) then
        c.s ← a.s ^ b.s
        c.t ← a.t
        c.e ← a.e
        c.f ← a.f
    elseif a.t=ZERO and b.t=INFINITY then
        c ← DEFAULTSNAN // Invalid
    elseif a.t=INFINITY and b.t=ZERO then
        c ← DEFAULTSNAN // Invalid
    elseif a.t=ZERO or b.t=ZERO then
        c.s ← a.s ^ b.s
        c.t ← ZERO
    else
        assert FALSE // should have covered al the cases above
    endif
enddef

def c ← fdiv(a,b) as
    if a.t=NORM and b.t=NORM then
        c.s ← a.s ^ b.s
        c.t ← NORM
        c.e ← a.e - b.e + 256
        c.f ← (a.f || 0^256) / b.f
    // priority is given to b operand for NaN propagation
    elseif (b.t=SNAN) or (b.t=QNAN) then
        c.s ← a.s ^ b.s
        c.t ← b.t
        c.e ← b.e
```

```
            c.f ← b.f
      elseif (a.t=SNAN) or (a.t=QNAN) then
            c.s ← a.s ^ b.s
            c.t ← a.t
            c.e ← a.e
            c.f ← a.f
      elseif a.t=ZERO and b.t=ZERO then
            c ← DEFAULTSNAN // Invalid
      elseif a.t=INFINITY and b.t=INFINITY then
            c ← DEFAULTSNAN // Invalid
      elseif a.t=ZERO then
            c.s ← a.s ^ b.s
            c.t ← ZERO
      elseif a.t=INFINITY then
            c.s ← a.s ^ b.s
            c.t ← INFINITY
      else
            assert FALSE // should have covered al the cases above
      endif
enddef

def msb ← findmsb(a) as
      MAXF ← 2^18 // Largest possible f value after matrix multiply
      for j ← 0 to MAXF
            if a_{MAXF-1..j} = (0^{MAXF-1-j} || 1) then
                  msb ← j
            endif
      endfor
enddef

def ai ← PackF(prec,a,round) as
      case a.t of
            NORM:
                  msb ← findmsb(a.f)
                  rn ← msb-1-fbits(prec) // lsb for normal
                  rdn ← -ebias(prec)-a.e-1-fbits(prec) // lsb if a denormal
                  rb ← (rn > rdn) ? rn : rdn
                  if rb ≤ 0 then
                        aifr ← a.f_{msb-1..0} || 0^{-rb}
                        eadj ← 0
                  else
                        case round of
                              C:
                                    s ← 0^{msb-rb} || (~a.s)^{rb}
                              F:
                                    s ← 0^{msb-rb} || (a.s)^{rb}
                              N, NONE:
                                    s ← 0^{msb-rb} || ~a.f_{rb} || a.f_{rb}^{rb-1}
                              X:
                                    if a.f_{rb-1..0} ≠ 0 then
                                          raise FloatingPointArithmetic // Inexact
                                    endif
                                    s ← 0
                              Z:
```

```
                    s ← 0
            endcase
            v ← (0 || a.f_{msb..0}) + (0 || s)
            if v_{msb} = 1 then
                aifr ← v_{msb-1..rb}
                eadj ← 0
            else
                aifr ← 0^{fbits(prec)}
                eadj ← 1
            endif
        endif
        aien ← a.e + msb - 1 + eadj + rbias(prec)
        if aien ≤ 0 then
            if round = NONE then
                ai ← a.s || 0^{ebits(prec)} || aifr
            else
                raise FloatingPointArithmetic //Underflow
            endif
        elseif aien ≥ 1^{ebits(prec)} then
            if round = NONE then
                //default: round-to-nearest overflow handling
                ai ← a.s || 1^{ebits(prec)} || 0^{fbits(prec)}
            else
                raise FloatingPointArithmetic //Underflow
            endif
        else
            ai ← a.s || aien_{ebits(prec)-1..0} || aifr
        endif
    SNAN:
        if round ≠ NONE then
            raise FloatingPointArithmetic //Invalid
        endif
        if -a.e < fbits(prec) then
            ai ← a.s || 1^{ebits(prec)} || a.f_{-a.e-1..0} || 0^{fbits(prec)+a.e}
        else
            lsb ← a.f_{-a.e-1-fbits(prec)+1..0} ≠ 0
            ai ← a.s || 1^{ebits(prec)} || a.f_{-a.e-1..-a.e-1-fbits(prec)+2} || lsb
        endif
    QNAN:
        if -a.e < fbits(prec) then
            ai ← a.s || 1^{ebits(prec)} || a.f_{-a.e-1..0} || 0^{fbits(prec)+a.e}
        else
            lsb ← a.f_{-a.e-1-fbits(prec)+1..0} ≠ 0
            ai ← a.s || 1^{ebits(prec)} || a.f_{-a.e-1..-a.e-1-fbits(prec)+2} || lsb
        endif
    ZERO:
        ai ← a.s || 0^{ebits(prec)} || 0^{fbits(prec)}
    INFINITY:
        ai ← a.s || 1^{ebits(prec)} || 0^{fbits(prec)}
    endcase
enddef
```

```
def ai ← fsinkr(prec, a, round) as
    case a.t of
        NORM:
            msb ← findmsb(a.f)
            rb ← -a.e
            if rb ≤ 0 then
                aifr ← a.f_msb..0 || 0^-rb
                aims ← msb - rb
            else
                case round of
                    C, C.D:
                        s ← 0^msb-rb || (-ai.s)^rb
                    F, F.D:
                        s ← 0^msb-rb || (ai.s)^rb
                    N, NONE:
                        s ← 0^msb-rb || -ai.f_rb || ai.f_rb^rb-1
                    X:
                        if ai.f_rb-1..0 ≠ 0 then
                            raise FloatingPointArithmetic // Inexact
                        endif
                        s ← 0
                    Z, Z.D:
                        s ← 0
                endcase
                v ← (0||a.f_msb..0) + (0||s)
                if v_msb = 1 then
                    aims ← msb + 1 - rb
                else
                    aims ← msb - rb
                endif
                aifr ← v_aims..rb
            endif
            if aims > prec then
                case round of
                    C.D, F.D, NONE, Z.D:
                        ai ← a.s || (-as)^prec-1
                    C, F, N, X, Z:
                        raise FloatingPointArithmetic // Overflow
                endcase
            elseif a.s = 0 then
                ai ← aifr
            else
                ai ← -aifr
            endif
        ZERO:
            ai ← 0^prec
        SNAN, QNAN:
            case round of
                C.D, F.D, NONE, Z.D:
                    ai ← 0^prec
                C, F, N, X, Z:
                    raise FloatingPointArithmetic // Invalid
            endcase
        INFINITY:
```

17

```
                    case round of
                         C.D F.D, NONE, Z.D:
                              ai ← a.s || (-a)^prec-1
                         C, F, N, X, Z:
                              raise FloatingPointArithmetic // Invalid
                    endcase
          endcase
enddef

def c ← frecrest(a) as
     b.s ← 0
     b.t ← NORM
     b.e ← 0
     b.f ← 1
     c ← fest(fdiv(b,a))
enddef

def c ← frsqrest(a) as
     b.s ← 0
     b.t ← NORM
     b.e ← 0
     b.f ← 1
     c ← fest(fsqr(fdiv(b,a)))
enddef

def c ← fest(a) as
     if (a.t=NORM) then
          msb ← findmsb(a.f)
          a.e ← a.e + msb - 13
          a.f ← a.f_msb..msb-12 || 1
     else
          c ← a
     endif
enddef

def c ← fsqr(a) as
     if (a.t=NORM) and (a.s=0) then
          c.s ← 0
          c.t ← NORM
          if (a.e_0 = 1) then
               c.e ← (a.e-127) / 2
               c.f ← sqr(a.f || 0^127)
          else
               c.e ← (a.e-128) / 2
               c.f ← sqr(a.f || 0^128)
          endif
     elseif (a.t=SNAN) or (a.t=QNAN) or a.t=ZERO or ((a.t=INFINITY) and (a.s=0)) then
          c ← a
     elseif ((a.t=NORM) or (a.t=INFINITY)) and (a.s=1) then
          c ← DEFAULTSNAN // Invalid
     else
          assert FALSE // should have covered al the cases above
     endif
enddef
```

Digital Signal Processing

The Zeus processor provides a set of operations that maintain the fullest possible use of 128-bit data paths when operating on lower-precision fixed-point or floating-point vector values. These operations are useful for several application areas, including digital signal processing, image processing and synthetic graphics. The basic goal of these operations is to accelerate the performance of algorithms that exhibit the following characteristics:

Low-precision arithmetic

The operands and intermediate results are fixed-point values represented in no greater than 64 bit precision. For floating-point arithmetic, operands and intermediate results are of 16, 32, or 64 bit precision.

The fixed-point arithmetic operations include add, subtract, multiply, divide, shifts, and set on compare.

The use of fixed-point arithmetic permits various forms of operation reordering that are not permitted in floating-point arithmetic. Specifically, commutativity and associativity, and distribution identities can be used to reorder operations. Compilers can evaluate operations to determine what intermediate precision is required to get the specified arithmetic result.

Zeus supports several levels of precision, as well as operations to convert between these different levels. These precision levels are always powers of two, and are explicitly specified in the operation code.

When specified, add, subtract, and shift operations may cause a fixed-point arithmetic exception to occur on resulting conditions such as signed or unsigned overflow. The fixed-point arithmetic exception may also be invoked upon a signed or unsigned comparison.

Sequential access to data

The algorithms are or can be expressed as operations on sequentially ordered items in memory. Scatter-gather memory access or sparse-matrix techniques are not required.

Where an index variable is used with a multiplier, such multipliers must be powers of two. When the index is of the form: nx+k, the value of n must be a power of two, and the values referenced should have k include the majority of values in the range 0..n-1. A negative multiplier may also be used.

Vectorizable operations

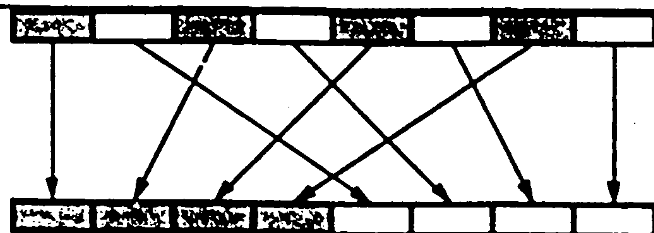
The operations performed on these sequentially ordered items are identical and independent. Conditional operations are either rewritten to use Boolean variables or masking, or the compiler is permitted to convert the code into such a form.

Data-handling Operations

The characteristics of these algorithms include sequential access to data, which permit the use of the normal load and store operations to reference the data. Octlet and hexlet loads and stores reference several sequential items of data, the number depending on the operand precision.

The discussion of these operations is independent of byte ordering, though the ordering of bit fields within octlets and hexlets must be consistent with the ordering used for bytes. Specifically, if big-endian byte ordering is used for the loads and stores, the figures below should assume that index values increase from left to right, and for little-endian byte ordering, the index values increase from right to left. For this reason, the figures indicate different index values with different shades, rather than numbering.

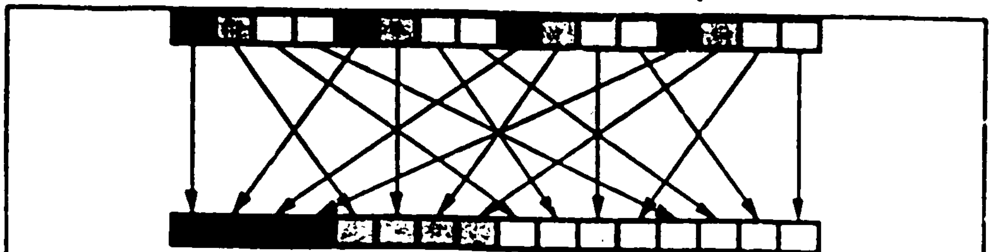
When an index of the nx+k form is used in array operands, where n is a power of 2, data memory sequentially loaded contains elements useful for separate operands. The "shuffle" instruction divides a triclet of data up into two hexlets, with alternate bit fields of the source triclet grouped together into the two results. An immediate field, h, in the instruction specifies which of the two regrouped hexlets to select for the result. For example, two X.SHUFFLE.256 rd=rc,rb,32,128,h operations rearrange the source triclet (c,b) into two hexlets as follows:



32-bit 2-way deal

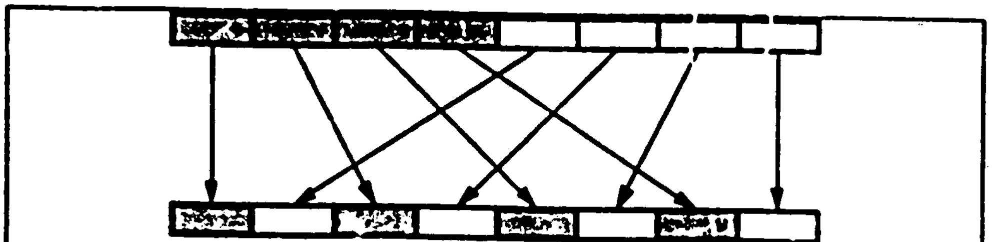
In the shuffle operation, two hexlet registers specify the source triclet, and one of the two result hexlets are specified as hexlet register.

The example above directly applies to the case where n is 2. When n is larger, shuffle operations can be used to further subdivide the sequential stream. For example, when n is 4, we need to deal out 4 sets of doublet operands, as shown in the figure below:[2]



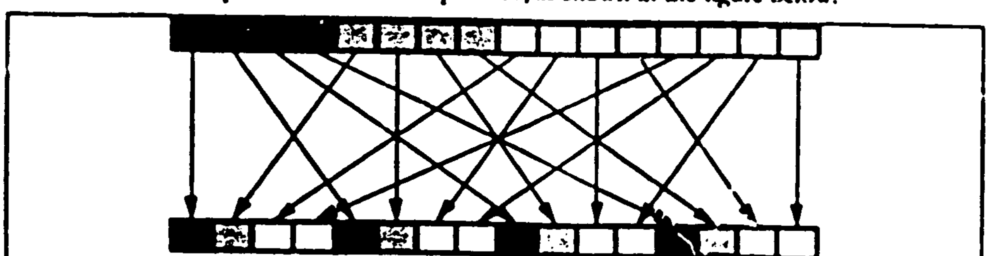
16-bit 4-way deal

When an array result of computation is accessed with an index of the form nx+k, for n a power of 2, the reverse of the "deal" operation needs to be performed on vectors of results to interleave them for storage in sequential order. The "shuffle" operation interleaves the bit fields of two octlets of results into a single hexlet. For example a X.SHUFFLE.16 operation combines two octlets of doublet fields into a hexlet as follows:



16-bit 2-way shuffle

For larger values of n, a series of shuffle operations can be used to combine additional sets of fields, similarly to the mechanism used for the deal operations. For example, when n is 4, we need to shuffle up 4 sets of doublet operands, as shown in the figure below:[3]



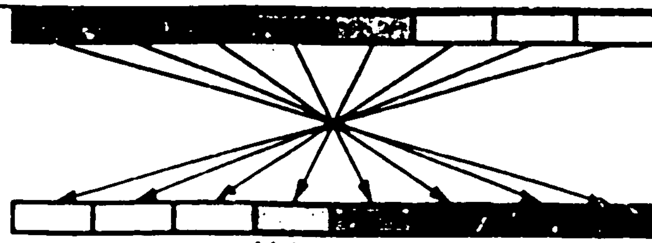
16-bit 4-way shuffle

When the index of a source array operand or a destination array result is negated, or in other words, if of the form nx+k where n is negative, the elements of the array must be arranged

[2] An example of the use of a four-way deal can be found in the appendix: Digital Signal Processing Applications: Conversion of Color to Monochrome

[3] An example of the use of a four-way shuffle can be found in the appendix: Digital Signal Processing Applications: Conversion of Monochrome to Color

in reverse order. The "swizzle" operation can reverse the order of the bit fields in a hexlet. For example, a X.SWIZZLE rd=rc,127,112 operation reverses the doublets within a hexlet:

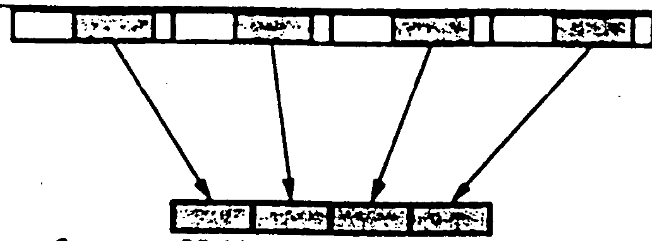


16-bit reverse

In some cases, it is desirable to use a group instruction in which one or more operands is a single value, not an array. The "swizzle" operation can also copy operands to multiple locations within a hexlet. For example, a X.SWIZZLE 15,0 operation copies the low-order 16 bits to each double within a hexlet.

Variations of the deal and shuffle operations are also useful for converting from one precision to another. This may be required if one operand is represented in a different precision than another operand or the result, or if computation must be performed with intermediate precision greater than that of the operands, such as when using an integer multiply.

When converting from a higher precision to a lower precision, specifically when halving the precision of a hexlet of bit fields, half of the data must be discarded, and the bit fields packed together. The "compress" operation is a variant of the "deal" operation, in which the operand is a hexlet, and the result is an octlet. An arbitrary half-sized sub-field of each bit field can be selected to appear in the result. For example, a selection of bits 19..4 of each quadlet in a hexlet is performed by the X.COMPRESS rd=rc,16,4 operation:

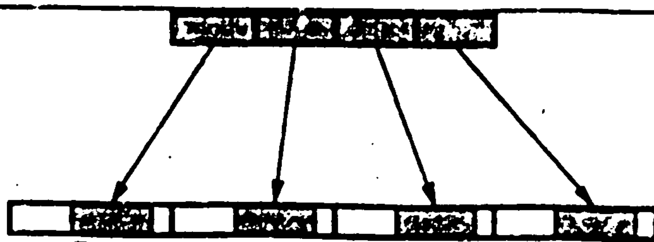


Compress 32 bits to 16, with 4-bit right shift

When converting from lower precision to higher-precision, specifically when doubling the precision of an octlet of bit fields, one of several techniques can be used, either multiply, expand, or shuffle. Each has certain useful properties. In the discussion below, m is the precision of the source operand.

The multiply operation, described in detail below, automatically doubles the precision of the result, so multiplication by a constant vector will simultaneously double the precision of the operand and multiply by a constant that can be represented in m bits.

An operand can be doubled in precision and shifted left with the "expand" operation, which is essentially the reverse of the "compress" operation. For example the X.EXPAND rd=rc,16,4 expands from 16 bits to 32, and shifts 4 bits left:



Expand 16 bits to 32, with 4-bit left shift

The "shuffle" operation can double the precision of an operand and multiply it by 1 (unsigned only), 2^m or 2^m+1, by specifying the sources of the shuffle operation to be a zeroed register and the source operand, the source operand and zero, or both to be the source operand. When multiplying by 2m, a constant can be freely added to the source operand by specifying the constant as the right operand to the shuffle.

Arithmetic Operations

The characteristics of the algorithms that affect the arithmetic operations most directly are low-precision arithmetic, and vectorizable operations. The fixed-point arithmetic operations provided are most of the functions provided in the standard integer unit, except for those that check conditions. These functions include add, subtract, bitwise Boolean operations, shift, set on condition, and multiply, in forms that take packed sets of bit fields of a specified size as operands. The floating-point arithmetic operations provided are as complete as the scalar floating-point arithmetic set. The result is generally a packed set of bit fields of the same size as the operands, except that the fixed-point multiply function intrinsically doubles the precision of the bit field.

Conditional operations are provided only in the sense that the set on condition operations can be used to construct bit masks that can select between alternate vector expressions, using the bitwise Boolean operations. All instructions operate over the entire octlet or hexlet operands, and produce a hexlet result. The sizes of the bit fields supported are always powers of two.

Galois Field Operations

Zeus provides a general software solution to the most common operations required for Galois Field arithmetic. The instructions provided include a polynomial multiply, with the polynomial specified as one register operand. This instruction can be used to perform CRC generation and checking, Reed-Solomon code generation and checking, and spread-spectrum encoding and decoding.

20

Software Conventions

The following section describes software conventions that are to be employed at software module boundaries, in order to permit the combination of separately compiled code and to provide standard interfaces between application, library and system software. Register usage and procedure call conventions may be modified, simplified or optimized when a single compilation encloses procedures within a compilation unit so that the procedures have no external interfaces. For example, internal procedures may permit a greater number of register-passed parameters, or have registers allocated to avoid the need to save registers at procedure boundaries, or may use a single stack or data pointer allocation to suffice for more than one level of procedure call.

Register Usage

All Zeus registers are identical and general-purpose; there is no dedicated zero-valued register, and no dedicated floating-point registers. However, some procedure-call-oriented instructions imply usage of registers zero (0) and one (1) in a manner consistent with the conventions described below. By software convention, the non-specific general registers are used in more specific ways.

register number	assembler names	usage	how saved
0	lp, r0	link pointer	caller
1	dp, r1	data pointer	caller
2-9	r2-r9	parameters	caller
10-31	r10-r31	temporary	caller
32-61	r32-r61	saved	callee
62	fp, r62	frame pointer	callee
63	sp, r63	stack pointer	callee

register usage

At a procedure call boundary, registers are saved either by the caller or callee procedure, which provides a mechanism for leaf procedures to avoid needing to save registers. Compilers may choose to allocate variables into caller or callee saved registers depending on how their lifetimes overlap with procedure calls.

Procedure Calling Conventions

Procedure parameters are normally allocated in registers, starting from register 2 up to register 9. These registers hold up to 8 parameters, which may each be of any size from one byte to sixteen bytes (hexlet), including floating-point and small structure parameters. Additional parameters are passed in memory, allocated on the stack. For C procedures which use varargs.h or stdarg.h and pass parameters to further procedures, the compilers must leave room in the stack memory allocation to save registers 2 through 9 into memory contiguously with the additional stack memory parameters, so that procedures such as _doprnt can refer to the parameters as an array.

Procedure return values are also allocated in registers, starting from register 2 up to register 9. Larger values are passed in memory, allocated on the stack.

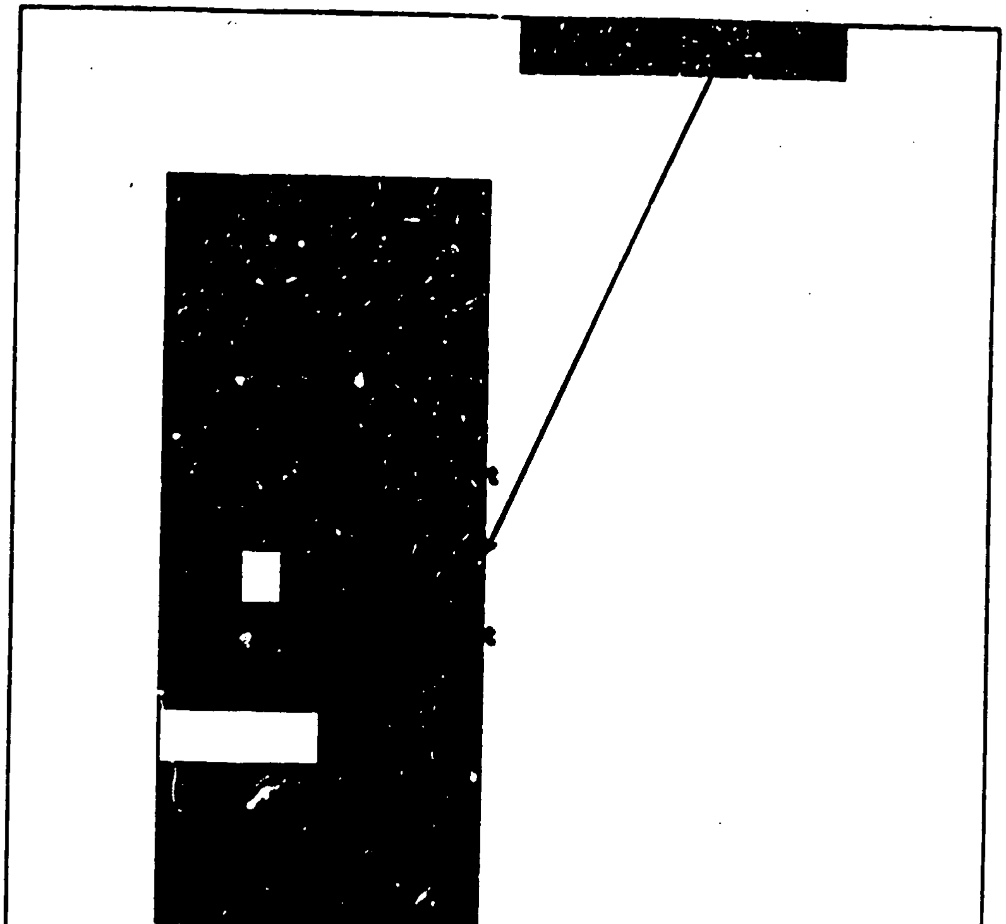
There are several pointers maintained in registers for the procedure calling conventions: lp, sp, dp, fp.

The lp register contains the address to which the callee should return to at the conclusion of the procedure. If the procedure is also a caller, the lp register will need to be saved on the stack, once, before any procedure call, and restored, once, after all procedure calls. The procedure returns with a branch instruction, specifying the lp register.

The sp register is used to form addresses to save parameter and other registers, maintain local variables, i.e., data that is allocated as a LIFO stack. For procedures that require a stack, normally a single allocation is performed, which allocates space for input parameters, local variables, saved registers, and output parameters all at once. The sp register is always hexlet aligned.

The dp register is used to address pointers, literals and static variables for the procedure. The dp register points to a small (approximately 4096-entry) array of pointers, literals, and statically-allocated variables, which is used locally to the procedure. The uses of the dp register are similar to the use of the gp register on a Mips R-series processor, except that each procedure may have a different value, which expands the space addressable by small offsets from this pointer. This is an important distinction, as the offset field of Zeus load and store instructions are only 12 bits. The compiler may use additional registers and/or indirect pointers to address larger regions for a single procedure. The compiler may also share a single dp register value between procedures which are compiled as a single unit (including procedures which are externally callable), eliminating the need to save, modify and restore the dp register for calls between procedures which share the same dp register value.

Load- and store- immediate-aligned instructions, specifying the dp register as the base register, are generally used to obtain values from the dp region. These instructions shift the immediate value by the logarithm of the size of the operand, so loads and stores of large operands may reach farther from the dp register than of small operands. The size of the addressable region is maximized if the elements to be placed in the dp region are sorted according to size, with the smallest elements placed closest to the dp base. At points where the size changes, appropriate padding is added to keep elements aligned to memory boundaries matching the size of the elements. Using this technique, the maximum size of the dp region is always at least 4096 items, and may be larger when the dp area is composed of a mixture of data sizes.



Alignment within dp region

The dp register mechanism also permits code to be shared, with each static instance of the dp region assigned to a different address in memory. In conjunction with position-independent or pc-relative branches, this allows library code to be dynamically relocated and shared between processes.

To implement an inter-module (separately compiled) procedure call, the lp register is loaded with the entry point of the procedure, and the dp register is loaded with the value of the dp register required for the procedure. These two values are located adjacent to each other as a pair of octlet quantities in the dp region for the calling procedure. For a statically-linked inter-module procedure call, the linker fills in the values at link time. However, this mechanism also provides for dynamic linking, by initially filling in the lp and dp fields in the data structure to invoke the dynamic linker. The dynamic linker can use the contents of the lp and/or dp registers to determine the identity of the caller and callee, to find the location to fill in the pointers and resume execution. Specifically, the lp value is initially set to point to an entry point in the dynamic linker, and the dp value is set to point to itself: the location of the lp and dp values in the dp region of the calling procedure. The identity of the procedure

can be discovered from a string following the dp pointer, or a separate table, indexed by the dp pointer.

The fp register is used to address the stack frame when the stack size varies during execution of a procedure, such as when using the GNU C alloca function. When the stack size can be determined at compile time, the sp register is used to address the stack frame and the fp register may be used for any other general purpose as a callee-saved register.

Typical static-linked, intra-module calling sequence:

caller (non-leaf):

```
caller:  AADDI          sp@-size      // allocate caller stack frame
         S.I.64.A       lp,sp,off     // save original lp register
         ... (callee using same dp as caller)
         B.LINK.I       callee
         ...
         ... (callee using same dp as caller)
         B.LINK.I       callee
         ...
         L.I.64.A       lp=sp,off     // restore original lp register
         AADDI          sp@size       // deallocate caller stack frame
         B              lp            // return
```

callee (leaf):

```
callee:  ... (code using dp)
         B              lp            // return
```

Procedures that are compiled together may share a common data region, in which case there is no need to save, load, and restore the dp region in the callee, assuming that the callee does not modify the dp register. The pc-relative addressing of the B.LINK.I instruction permits the code region to be position-independent.

Minimum static-linked, intra-module calling sequence:

caller (non-leaf):

```
caller:  ACOPY          r31=lp        // save original lp register
         ... (callee using same dp as caller)
         B.LINK.I       callee
         ...
         ... (callee using same dp as caller)
         B.LINK.I       callee
         ...
         B              r31           // return
```

callee (leaf):

```
callee:  ... (code using dp, r31 unused)
         B              lp            // return
```

When all the callee procedures are intra-module, the stack frame may also be eliminated from the caller procedure by using "temporary" caller save registers not utilized by the callee leaf procedures. In addition to the lp value indicated above, this usage may include other values and variables that live in the caller procedure across callee procedure calls.

Typical dynamic-linked, inter-module calling sequence:

caller (non-leaf):

```
caller:   AADDI          sp@-size      // allocate caller stack frame
          S.I.64.A       lp,sp,off     // save original lp register
          S.I.64.A       dp,sp,off     // save original dp register
          ... (code using dp)
          L.I.64.A       lp=dp,off     // load lp
          L.I.64.A       dp=dp,off     // load dp
          B.LINK         lp=lp         // invoke callee procedure
          L.I.64.A       dp=sp,off     // restore dp register from stack
          ... (code using dp)
          L.I.64.A       lp=sp,off     // restore original lp register
          AADDI          sp=size       // deallocate caller stack frame
          B              lp            // return
```

callee (leaf):

```
callee:   ... (code using dp)
          B              lp            // return
```

The load instruction is required in the caller following the procedure call to restore the dp register. A second load instruction also restores the lp register, which may be located at any point between the last procedure call and the branch instruction which returns from the procedure.

System and Privileged Library Calls

It is an objective to make calls to system facilities and privileged libraries as similar as possible to normal procedure calls as described above. Rather than invoke system calls as an exception, which involves significant latency and complication, we prefer to use a modified procedure call in which the process privilege level is quietly raised to the required level. To provide this mechanism safely, interaction with the virtual memory system is required.

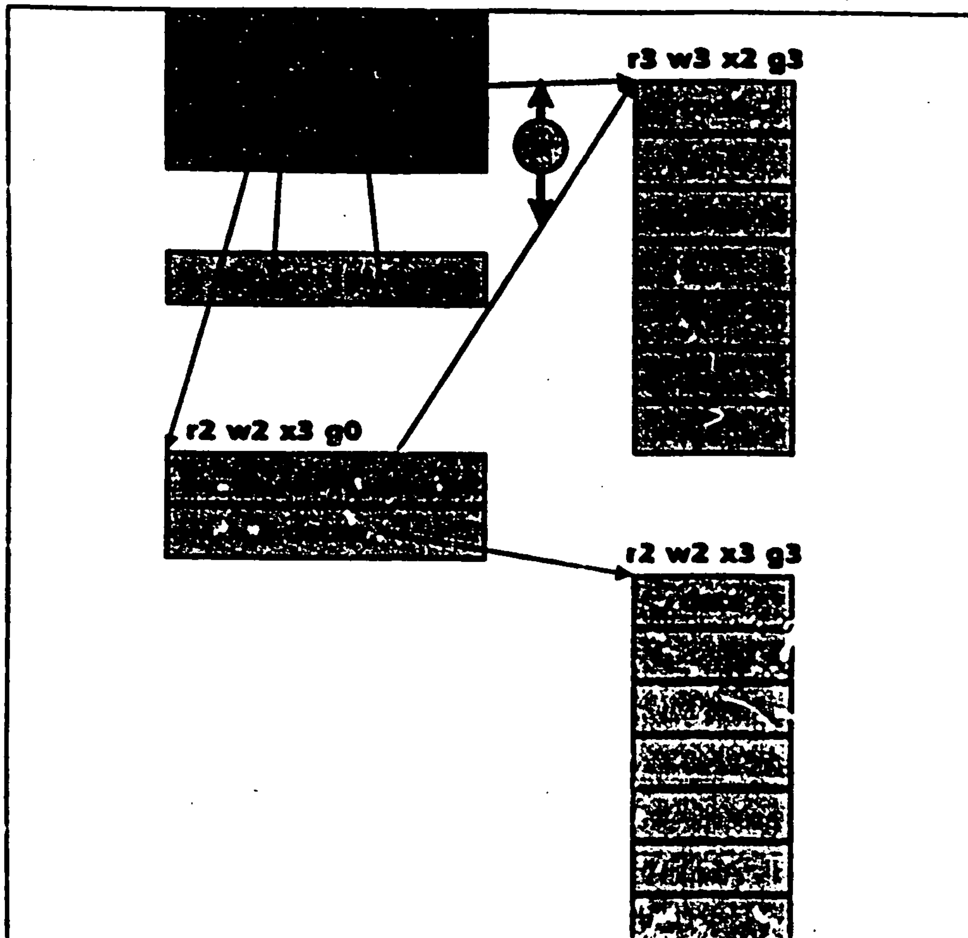
Such a procedure must not be entered from anywhere other than its legitimate entry point, to prohibit entering a procedure after the point at which security checks are performed or with invalid register contents, otherwise the access to a higher privilege level can lead to a security violation. In addition, the procedure generally must have access to memory data, for which addresses must be produced by the privileged code. To facilitate generating these addresses, the branch-gateway instruction allows the privileged code procedure to rely the fact that a single register has been verified to contain a pointer to a valid memory region.

The branch-gateway instruction ensures both that the procedure is invoked at a proper entry point, and that other registers such as the data pointer and stack pointer can be properly set. To ensure this, the branch-gateway instruction retrieves a "gateway" directly from the protected virtual memory space. The gateway contains the virtual address of the entry point of the procedure and the target privilege level. A gateway can only exist in regions of the virtual address space designated to contain them, and can only be used to access privilege levels at or below the privilege level at which the memory region can be written to ensure that a gateway cannot be forged.

The branch-gateway instruction ensures that register 1 (dp) contains a valid pointer to the gateway for this target code address by comparing the contents of register 0 (lp) against the

gateway retrieved from memory and causing an exception trap if they do not match. By ensuring that register 1 points to the gateway, auxiliary information, such as the data pointer and stack pointer can be set by loading values located by the contents of register 1. For example, the eight bytes following the gateway may be used as a pointer to a data region for the procedure.

Before executing the branch-gateway instruction, register 1 must be set to point at the gateway, and register 0 must be set to the address of the target code address plus the desired privilege level. A "L.I.64.L.A r0=r1,0" instruction is one way to set register 0, if register 1 has already been set, but any means of getting the correct value into register 0 is permissible.



Gateway with pointers to code and data spaces

Similarly, a return from a system or privileged routine involves a reduction of privilege. This need not be carefully controlled by architectural facilities, so a procedure may freely branch to a less-privileged code address. Normally, such a procedure restores the stack frame, then uses the branch-down instruction to return.

Typical dynamic-linked, inter-gateway calling sequence:

caller:

```
caller:  AADDI      sp@-size     // allocate caller stack frame
         S.I.64.A   lp,sp,off
         S.I.64.A   dp,sp,off
         ...
         L.I.64.A   lp=dp,off    // load lp
         L.I.64.A   dp=dp,off    // load dp
         B.GATE
         L.I.64.A   dp,sp,off
         ... (code using dp)
         L.I.64.A   lp=sp,off    // restore original lp register
         AADDI      sp=size      // deallocate caller stack frame
         B          lp           // return
```

callee (non-leaf):

```
callee:  L.I.64.A   dp=dp,off    // load dp with data pointer
         S.I.64.A   sp,dp,off
         L.I.64.A   sp=dp,off    // new stack pointer
         S.I.64.A   lp,sp,off
         S.I.64.A   dp,sp,off
         ... (using dp)
         L.I.64.A   dp,sp,off
         ... (code using dp)
         L.I.64.A   lp=sp,off    // restore original lp register
         L.I.64.A   sp=sp,off    // restore original sp register
         B.DOWN     lp
```

callee (leaf, no stack):

```
callee:  ... (using dp)
         B.DOWN     lp
```

It can be observed that the calling sequence is identical to that of the inter-module calling sequence shown above, except for the use of the B.GATE instruction instead of a B.LINK instruction. Indeed, if a B.GATE instruction is used when the privilege level in the lp register is not higher than the current privilege level, the B.GATE instruction performs an identical function to a B.LINK.

The callee, if it uses a stack for local variable allocation, cannot necessarily trust the value of the sp passed to it, as it can be forged. Similarly, any pointers which the callee provides should not be used directly unless it they are verified to point to regions which the callee should be permitted to address. This can be avoided by defining application programming interfaces (APIs) in which all values are passed and returned in registers, or by using a trusted, intermediate privilege wrapper routine to pass and return parameters. The method described below can also be used.

It can be useful to have highly privileged code call less-privileged routines. For example, a user may request that errors in a privileged routine be reported by invoking a user-supplied error-logging routine. To invoke the procedure, the privilege can be reduced via the branch-down instruction. The return from the procedure actually requires an increase in privilege, which must be carefully controlled. This is dealt with by placing the procedure call within a lower-privilege procedure wrapper, which uses the branch-gateway instruction to return to

the higher privilege region after the call through a secure re-entry point. Special care must be taken to ensure that the less-privileged routine is not permitted to gain unauthorized access by corruption of the stack or saved registers, such as by saving all registers and setting up a new stack frame (or restoring the original lower-privilege stack) that may be manipulated by the less-privileged routine. Finally, such a technique is vulnerable to an unprivileged routine attempting to use the re-entry point directly, so it may be appropriate to keep a privileged state variable which controls permission to enter at the re-entry point.

Instruction Scheduling

The next section describes detailed pipeline organization for Zeus, which has a significant influence on instruction scheduling. Here we will elaborate some general rules for effective scheduling by a compiler. Specific information on numbers of functional units, functional unit parallelism and latency is quite implementation-dependent, values indicated here are valid for Zeus's first implementation.

Separate Addressing from Execution

Zeus has separate function units to perform addressing operations (A, L, S, B instructions) from execution operations (G, X, E, W instructions). When possible, Zeus will execute all the addressing operations of an instruction stream, deferring execution of the execution operations until dependent load instructions are completed. Thus, the latency of the memory system is hidden, so long as addressing operations themselves do not need to wait for memory.

Software Pipeline

Instructions should generally be scheduled so that previous operations can be completed at the time of issue. When this is not possible, the processor inserts sufficient empty cycles to perform the instructions precisely - explicit no-operation instructions are not required.

Multiple Issue

Zeus can issue up to two addressing operations and up to two execution operations per cycle per thread. Considering functional unit parallelism, described below, as many of four instruction issues per cycle are possible per thread.

Functional Unit parallelism

Zeus has separate function units for several classes of execution operations. An A unit performs scalar add, subtract, boolean, and shift-add operations for addressing and branch calculations. The remaining functional units are execution resources, which perform operations subsequent to memory loads and which operate on values in a parallel, partitioned form. A G unit performs add, subtract, boolean, and shift-add operations. An X unit performs general shift operations. An E unit performs multiply and floating-point operations. A T unit performs table-look-up operations.

Each instruction uses one or more of these units, according to the table below.

Instruction	A	G	X	E	T
A	x				
B	x				
L	x				
S	x				
G		x			
X			x		
E			x	x	
W.TRANSLATE	x				x
W.MULMAT	x		x	x	
W.SWITCH	x		x		

Latency

The latency of each functional unit depends on what operation is performed in the unit, and where the result is used. The aggressive nature of the pipeline makes it difficult to characterize the latency of each operation with a single number. Because the addressing unit is decoupled from the execution unit, the latency of load operations is generally hidden, unles the result of a load instruction must be returned to the addressing unit. Store instructions must be able to compute the address to which the data is to be stored in the addressing unit, but the data will not be irrevocably stored until the data is available and it is valid to retire the store instruction. However, under certain conditions, data may be forwarded from a store instruction to subsequent load instructions, once the data is available.

The latency of each of these units, for the initial Zeus implementation is indicated below:

Unit	instruction	Latency rules
A	A	1 cycle
	L	Address operands must be ready to issue. 4 cycles to A unit, 0 to G, X, E, T units
	S	Address operands must be ready to issue. Store occurs when data is ready and instruction may be retired.
	B	Conditional branch operands may be provided from the A unit (64-bit values), or the G unit (128-bit values). 4 cycles for mispredicted branch
	W	Address operand must be ready to issue.
G	G	1 cycle
X	X, W.SWITCH	1 cycle for data operands, 2 cycles for shift amount or control operand
E	E, W.MULMAT	4 cycles
T	W.TRANSLATE	1 cycles

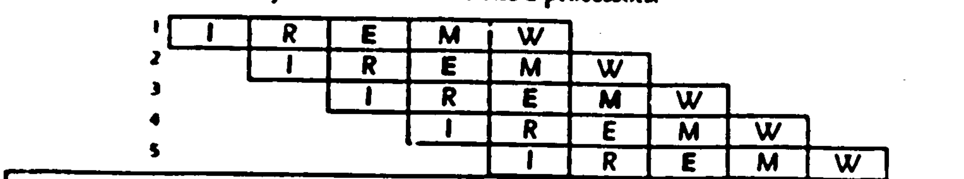
Pipeline Organization

Zeus performs all instructions *as if* executed one-by-one, in-order, with precise exceptions always available. Consequently, code that ignores the subsequent discussion of Zeus pipeline implementations will still perform correctly. However, the highest performance of the Zeus processor is achieved only by matching the ordering of instructions to the characteristics of the pipeline. In the following discussion, the general characteristics of all Zeus implementations precede discussion of specific choices for specific implementations.

Classical Pipeline Structures

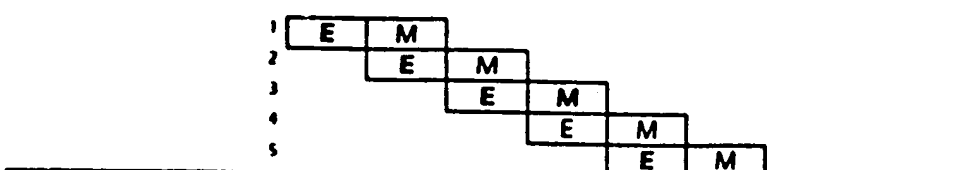
Pipelining in general refers to hardware structures that overlap various stages of execution of a series of instructions so that the time required to perform the series of instructions is less than the sum of the times required to perform each of the instructions separately. Additionally, pipelines carry to connotation of a collection of hardware structures which have a simple ordering and where each structure performs a specialized function.

The diagram below shows the timing of what has become a canonical pipeline structure for a simple RISC processor, with time on the horizontal axis increasing to the right, and successive instructions on the vertical axis going downward. The stages I, R, E, M, and W refer to units which perform instruction fetch, register file fetch, execution, data memory fetch, and register file write. The stages are aligned so that the result of the execution of an instruction may be used as the source of the execution of an immediately following instruction, as seen by the fact that the end of an E stage (bold in line 1) lines up with the beginning of the E stage (bold in line 2) immediately below. Also, it can be seen that the result of a load operation executing in stages E and M (bold in line 3) is not available in the immediately following instruction (line 4), but may be used two cycles later (line 5); this is the cause of the load delay slot seen on some RISC processors.



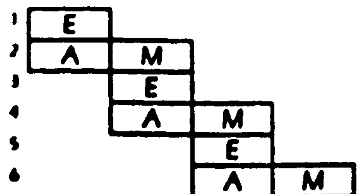
canonical pipeline

In the diagrams below, we simplify the diagrams somewhat by eliminating the pipe stages for instruction fetch, register file fetch, and register file write, which can be understood to precede and follow the portions of the pipelines diagrammed. The diagram above is shown again in this new format, showing that the canonical pipeline has very little overlap of the actual execution of instructions.



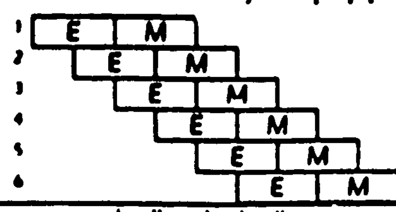
canonical pipeline

superscalar pipeline is one capable of simultaneously issuing two or more instructions which are independent, in that they can be executed in either order and separately, producing the same result as if they were executed serially. The diagram below shows a two-way superscalar processor, where one instruction may be a register-to-register operation (using stage E) and the other may be a register-to-register operation (using stage A) or a memory load or store (using stages A and M).



superscalar pipeline

A superpipelined pipeline is one capable is issuing simple instructions frequently enough that the result of a simple instruction must be independent of the immediately following one or more instructions. The diagram below shows a two-cycle superpipelined implementation:



superpipelined pipeline

In the diagrams below, pipeline stages are labelled with the type of instruction that may be performed by that stage. The position of the stage further identifies the function of that stage, as for example a load operation may require several L stages to complete the instruction.

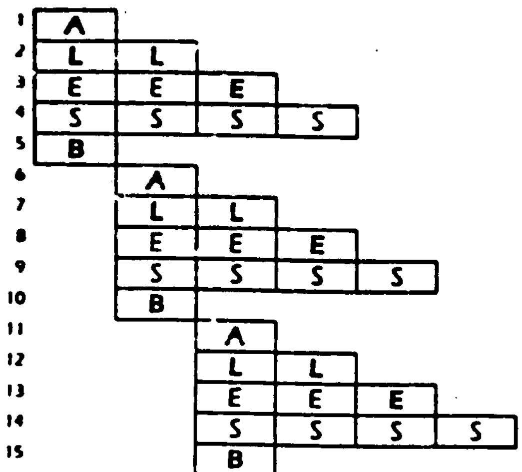
Superstring Pipeline

Zeus architecture provides for implementations designed to fetch and execute several instructions in each clock cycle. For a particular ordering of instruction types, one instruction of each type may be issued in a single clock cycle. The ordering required is A, L, E, S, B; in other words, a register-to-register address calculation, a memory load, a register-to-register data calculation, a memory store, and a branch. Because of the organization of the pipeline, each of these instructions may be serially dependent. Instructions of type E include the fixed-point execute-phase instructions as well as floating-point and digital signal processing instructions. We call this form of pipeline organization "superstring,"[4] because of the ability to issue a string of dependent instructions in a single clock cycle, as distinguished

[4]Readers with a background in theoretical physics may have seen this term in an other, unrelated, context.

from superscalar or superpipelined organizations, which can only issue sets of independent instructions.

These instructions take from one to four cycles of latency to execute, and a branch prediction mechanism is used to keep the pipeline filled. The diagram below shows a box for the interval between issue of each instruction and the completion. Bold letters mark the critical latency paths of the instructions, that is, the periods between the required availability of the source registers and the earliest availability of the result registers. The A-L critical latency path is a special case, in which the result of the A instruction may be used as the base register of the L instruction without penalty. E instructions may require additional cycles of latency for certain operations, such as fixed-point multiply and divide, floating-point and digital signal processing operations.

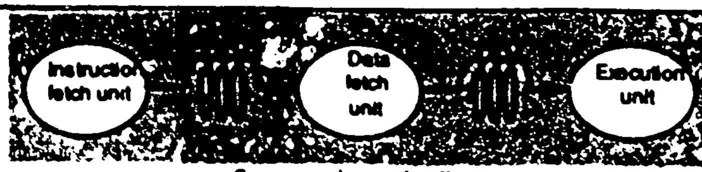


Superstring pipeline

Superspring Pipeline

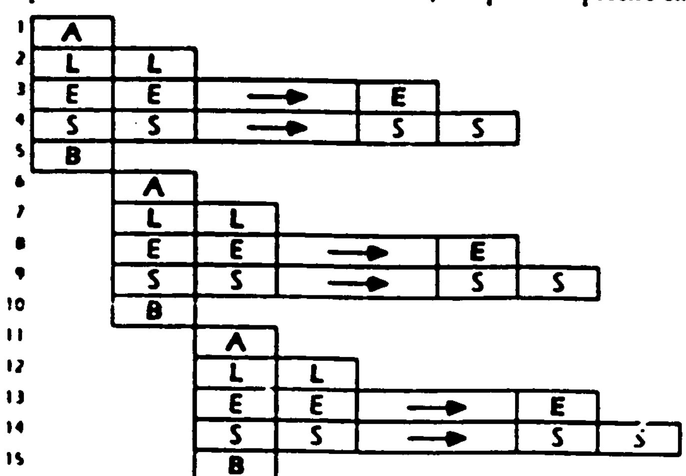
Zeus architecture provides an additional refinement to the organization defined above, in which the time permitted by the pipeline to service load operations may be flexibly extended. Thus, the front of the pipeline, in which A, L and B type instructions are handled, is decoupled from the back of the pipeline, in which E, and S type instructions are handled. This decoupling occurs at the point at which the data cache and its backing memory is referenced; similarly, a FIFO that is filled by the instruction fetch unit decouples instruction cache references from the front of the pipeline shown above. The depth of the FIFO structures is implementation-dependent, i.e. not fixed by the architecture.

The diagram below indicates why we call this pipeline organization feature "superspring," an extension of our superstring organization.



Superspring pipeline

With the super-spring organization, the latency of load instructions can be hidden, as execute instructions are deferred until the results of the load are available. Nevertheless, the execution unit still processes instructions in normal order, and provides precise exceptions.



Superspring pipeline

Superthread Pipeline

This technique is not employed in the initial Zeus implementation, though it was present in an earlier prototype implementation.

A difficulty of superpipelining is that dependent operations must be separated by the latency of the pipeline, and for highly pipelined machines, the latency of simple operations can be quite significant. The Zeus "superthread" pipeline provides for very highly pipelined implementations by alternating execution of two or more independent threads. In this context, a thread is the state required to maintain an independent execution; the architectural state required is that of the register file contents, program counter, privilege level, local TB, and when required, exception status. Ensuring that only one thread may handle an exception at one time may minimize the latter state, exception status. In order to ensure that all threads make reasonable forward progress, several of the machine resources must be scheduled fairly.

An example of a resource that is critical that it be fairly shared is the data memory/cache subsystem. In a prototype implementation, Zeus is able to perform a load operation only on every second cycle, and a store operation only on every fourth cycle. Zeus schedules these fixed timing resources fairly by using a round-robin schedule for a number of threads that is relatively prime to the resource reuse rates. For this implementation, five simultaneous threads of execution ensure that resources which may be used every two or four cycles are fairly shared by allowing the instructions which use those resources to be issued only on every second or fourth issue slot for that thread.

In the diagram below, the thread number which issues an instruction is indicated on each clock cycle, and below it, a list of which functional units may be used by that instruction. The diagram repeats every 20 cycles, so cycle 20 is similar to cycle 0, cycle 21 is similar to cycle 1, etc. This schedule ensures that no resource conflict occur between threads for these resources. Thread 0 may issue an E, L, S or B on cycle 0, but on its next opportunity, cycle 5, may only issue E or B, and on cycle 10 may issue E, L or B, and on cycle 15, may issue E or B.

cycle	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
thread	0	1	2	3	4	0	1	2	3	4	0	1	2	3	4	0	1	2	3	4
	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E
	L		L		L		L		L		L		L		L		L		L	
	S				S				S				S				S			
	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B

Superthread pipeline

When seen from the perspective of an individual thread, the resource use diagram looks similar to that of the collection. Thus an individual thread may use the load unit every two instructions, and the store unit every four instructions.

cycle	0	5	10	15	20	25	30	35	40	45	50	55	60	65	70	75	80	85	90	95
thread	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E
	L		L		L		L		L		L		L		L		L		L	
	S				S				S				S				S			
	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B

Superthread pipeline

A Zeus Superthread pipeline, with 5 simultaneous threads of execution, permits simple operations, such as register-to-register add (G.ADD), to take 5 cycles to complete, allowing for an extremely deeply pipelined implementation.

Simultaneous Multithreading

The intial Zeus implementation performs simultaneous multithreading among 4 threads. Each of the 4 threads share a common memory system, a common T unit. Pairs of threads share two G units, one X unit, and one E unit. Each thread individually has two A units. A fair allocation scheme balances access to the shared resources by the four threads.

Branch/fetch Prediction

Zeus does not have delayed branch instructions, and so relies upon branch or fetch prediction to keep the pipeline full around unconditional and conditional branch instructions. In the simplest form of branch prediction, as in Zeus's first implementation, a taken conditional backward (toward a lower address) branch predicts that a future execution of the same branch will be taken. More elaborate prediction may cache the source and target addresses of multiple branches, both conditional and unconditional, and both forward and reverse.

The hardware prediction mechanism is tuned for optimizing conditional branches that close loops or express frequent alternatives, and will generally require substantially more cycles when executing conditional branches whose outcome is not predominately taken or not-taken. For such cases of unpredictable conditional results, the use of code that avoids conditional branches in favor of the use of compare-set and multiplex instructions may result in greater performance.

Under some conditions, the above technique may not be applicable, for example if the conditional branch "guards" code which cannot be performed when the branch is taken. This may occur, for example, when a conditional branch tests for a valid (non-zero) pointer and the conditional code performs a load or store using the pointer. In these cases, the conditional branch has a small positive offset, but is unpredictable. A Zeus pipeline may handle this case as if the branch is always predicted to be not taken, with the recovery of a misprediction causing cancellation of the instructions which have already been issued but not completed which would be skipped over by the taken conditional branch. This "conditional-skip" optimization is performed by the initial Zeus implementation and requires no specific architectural feature to access or implement.

A Zeus pipeline may also perform "branch-return" optimization, in which a branch-link instruction saves a branch target address that is used to predict the target of the next returning branch instruction. This optimization may be implemented with a depth of one (only one return address kept), or as a stack of finite depth, where a branch and link pushes onto the stack, and a branch-register pops from the stack. This optimization can eliminate the misprediction cost of simple procedure calls, as the calling branch is susceptible to hardware prediction, and the returning branch is predictable by the branch-return optimization. Like the conditional-skip optimization described above, this feature is performed by the initial Zeus implementation and requires no specific architectural feature to access or implement.

Zeus implements two related instructions that can eliminate or reduce branch delays for conditional loops, conditional branches, and computed branches. The "branch-hint" instruction has no effect on architectural state, but informs the instruction fetch unit of a potential future branch instruction, giving the addresses of both the branch instruction and of the branch target. The two forms of the instruction specify the branch instruction address relative to the current address as an immediate field, and one form (branch-hint-immediate) specifies the branch target address relative to the current address as an immediate field, and the other (branch-hint) specifies the branch target address from a general register. The branch-hint-immediate instruction is generally used to give advance notice to the instruction

fetch unit of a branch-conditional instruction, so that instructions at the target of the branch can be fetched in advance of the branch-conditional instruction reaching the execution pipeline. Placing the branch hint as early as possible, and at a point where the extra instruction will not reduce the execution rate optimizes performance. In other words, an optimizing compiler should insert the branch-hint instruction as early as possible in the basic block where the parcel will contain at most one other "front-end" instruction.

Additional Load and Execute Resources

Studies of the dynamic distribution of Zeus instructions on various benchmark suites indicate that the most frequently-issued instruction classes are load instructions and execute instructions. In a high-performance Zeus implementation, it is advantageous to consider execution pipelines in which the ability to target the machine resources toward issuing load and execute instructions is increased.

One of the means to increase the ability to issue execute-class instructions is to provide the means to issue two execute instructions in a single-issue string. The execution unit actually requires several distinct resources, so by partitioning these resources, the issue capability can be increased without increasing the number of functional units, other than the increased register file read and write ports. The partitioning favored for the initial implementation places all instructions that involve shifting and shuffling in one execution unit, and all instructions that involve multiplication, including fixed-point and floating-point multiply and add in another unit. Resources used for implementing add, subtract, and bitwise logical operations may be duplicated, being modest in size compared to the shift and multiply units, or shared between the two units, as the operations have low-enough latency that two operations might be pipelined within a single issue cycle. These instructions must generally be independent, except perhaps that two simple add, subtract, or bitwise logical instructions may be performed dependently, if the resources for executing simple instructions are shared between the execution units.

One of the means to increase the ability to issue load-class instructions is to provide the means to issue two load instructions in a single-issue string. This would generally increase the resources required of the data fetch unit and the data cache, but a compensating solution is to steal the resources for the store instruction to execute the second load instruction. Thus, a single-issue string can then contain either two load instructions, or one load instruction and one store instruction, which uses the same register read ports and address computation resources as the basic 5-instruction string. This capability also may be employed to provide support for unaligned load and store instructions, where a single-issue string may contain as an alternative a single unaligned load or store instruction which uses the resources of the two load-class units in concert to accomplish the unaligned memory operation.

Result Forwarding

When temporally adjacent instructions are executed by separate resources, the results of the first instruction must generally be forwarded directly to the resource used to execute the second instruction, where the result replaces a value which may have been fetched from a register file. Such forwarding paths use significant resources. A Zeus implementation must

generally provide forwarding resources so that dependencies from earlier instructions within a string are immediately forwarded to later instructions, except between a first and second execution instruction as described above. In addition, when forwarding results from the execution units back to the data fetch unit, additional delay may be incurred.

Instruction Set

This section describes the instruction set in complete architectural detail. Operation codes are numerically defined by their position in the following operation code tables, and are referred to symbolically in the detailed instruction definitions. Entries that span more than one location in the table define the operation code identifier as the smallest value of all the locations spanned. The value of the symbol can be calculated from the sum of the legend values to the left and above the identifier.

Instructions that have great similarity and identical formats are grouped together. Starting on a new page, each category of instructions is named and introduced.

The Operation codes section lists each instruction by mnemonic that is defined on that page. A textual interpretation of each instruction is shown beside each mnemonic.

The Equivalences section lists additional instructions known to assemblers that are equivalent or special cases of base instructions, again with a textual interpretation of each instruction beside each mnemonic. Below the list, each equivalent instruction is defined, either in terms of a base instruction or another equivalent instruction. The symbol between the instruction and the definition has a particular meaning. If it is an arrow (← or →), it connects two mathematicaly equivalent operations, and the arrow direction indicates which form is preferred and produced in a reverse assembly. If the symbol is a (⇐), the form on the left is assembled into the form on the right solely for encoding purposes, and the form on the right is otherwise illegal in the assembler. The parameters in these definitions are formal; the names are solely for pattern-matching purposes, even though they may be suggestive of a particular meaning.

The Redundancies section lists instructions and operand values that may also be performed by other instructions in the instruction set. The symbol connecting the two forms is a (⇔), which indicates that the two forms are mathematically equivalent, both are legal, but the assembler does not transform one into the other.

The Selection section lists instructions and equivalences together in a tabular form that highlights the structure of the instruction mnemonics.

The Format section lists (1) the assembler format, (2) the C intrinsics format, (3) the bit-level instruction format, and (4) a definition of bit-level instruction format fields that are not a one-for-one match with named fields in the assembler format.

The Definition section gives a precise definition of each basic instruction.

The Exceptions section lists exceptions that may be caused by the execution of the instructions in this category.

Major Operation Codes

All instructions are 32 bits in size, and use the high order 8 bits to specify a major operation code.

31 … 24	23 … 0
major	other
8	24

The major field is filled with a value specified by the following table:[5]

MAJOR	0	32	64	96	128	160	192	224
0	ARES	BEF16	LI16L	SI16L		XDEPOSIT	EMULX	WMULMATXL
1	ADDI	BEF32	LI16B	SI16B	GADDI		EMULXU	WMULMATXB
2	ADDIO	BEF64	LI16AL	SI16AL	GADDIO		EMULXM	WMULMATXUL
3	ADDIUO	BEF128	LI16AB	SI16AB	GADDIUO		EMULXC	WMULMATXUB
4		BLGF16	LI32L	SI32L		XDEPOSITU	EMULADDX	WMULMATXML
5	ASUBI	BLGF32	LI32B	SI32B	GSUBI		EMULADDXU	WMULMATXMB
6	ASUBIO	BLGF64	LI32AL	SI32AL	GSUBIO		EMULADDXM	WMULMATXCL
7	ASUBIUO	BLGF128	LI32AB	SI32AB	GSUBIUO		EMULADDXC	WMULMATXCB
8	ASETEI	BLF16	LI64L	SI64L	GSETEI	XWITHDRAW	ECONXL	
9	ASETNEI	BLF32	LI64B	SI64B	GSETNEI		ECONXB	
10	ASETANDEI	BLF64	LI64AL	SI64AL	GSETANDEI		ECONXUL	
11	ASETANDNEI	BLF128	LI64AB	SI64AB	GSETANDNEI		ECONXUB	
12	ASETLI	BGEF16	LI128L	SI128L	GSETLI	XWITHDRAWU	ECONXML	
13	ASETGEI	BGEF32	LI128B	SI128B	GSETGEI		ECONXMB	
14	ASETLIU	BGEF64	LI128AL	SI128AL	GSETLIU		ECONXCL	
15	ASETGEIU	BGEF128	LI128AB	SI128AB	GSETGEIU		ECONXCB	
16	AANDI	BE	LUI16L	SASI64AL	GANDI	XDEPOSITM	ESCALADDF16	WMULMATFL
17	ANANDI	BNE	LUI16B	SASI64AB	GNANDI		ESCALADDF32	WMULMATFB
18	AORI	BANDE	LUI16AL	SCSI64AL	GORI		ESCALADDF64	WMULMATGL
19	ANORI	BANDNE	LUI16AB	SCSI64AB	GNORI		ESCALADDX	WMULMATGB
20	AXORI	BL	LUI32L	SMSI64AL	GXORI	XSWIZZLE	EMULG8	
21	AMUXI	BGE	LUI32B	SMSI64AB	GMUX		EMULG64	
22		BLU	LUI32AL	SMUXI64AL	GBOOLEAN		EMULX	
23		BGEU	LUI32AB	SMUXI64AB			EEXTRACT	
24	ACOPYI	BVF32	LUI64L		GCOPYI	XEXTRACT	EEXTRACTI	
25		BNVF32	LUI64B			XSELECT8	EEXTRACTIU	
26		BIF32	LUI64AL					WTABLEL
27		BNIF32	LUI64AB		G8		E8	WTABLEB
28		BI	LI8	SI8	G16	XSHUFFLE	E16	WSWITCHL
29		BLINK	LUI8		G32	XSHIFTI	E32	WSWITCHB
30		BHINTI			G64	XSHIFT	E64	WMINORL
31	AMINOR	BMINOR	LMINOR	SMINOR	G128		E128	WMINORB

major operation code field values

[5]Blank table entries cause the Reserved Instruction exception to occur.

Minor Operation Codes

For the major operation field values A.MINOR, B.MINOR, L.MINOR, S.MINOR, G.8, G.16, G.32, G.64, G.128, XSHIFTI, XSHIFT, E.8, E.16, E.32, E.64, E.128, W.MINOR.L and W.MINOR.B, the lowest-order six bits in the instruction specify a minor operation code:

31 24	23 6	5 0
major	other	minor
8	18	6

The minor field is filled with a value from one of the following tables:

A.MINOR	0	8	16	24	32	40	48	56
0		AAND	ASETE	ASETEF		ASHLI	ASHLIADD	
1	AADD	AXOR	ASETNE	ASETLGF				
2	AADDO	AOR	ASETANDE	ASETLF		ASHLIO		
3	AADDUO	AANDN	ASETANDNE	ASETGEF		ASHLIUO		
4		AORN	ASETL/LZ	ASETEF.X			ASHLISUB	
5	ASUB	AXNOR	ASETGE/GEZ	ASETLGF.X				
6	ASUBO	ANOR	ASETLU/GZ	ASETLF.X		ASHRI		
7	ASUBUO	ANAND	ASETGEU/LEZ	ASETGEF.X		ASHRIU		ACOM

minor operation code field values for A.MINOR

B.MINOR	0	8	16	24	32	40	48	56
0	B							
1	BLINK							
2	BHINT							
3	BDOWN							
4	BGATE							
5	BBACK							
6	BHALT							
7	BBARRIER							

minor operation code field values for B.MINOR

L.MINOR	0	8	16	24	32	40	48	56
0	L16L	L64L	LU16L	LU64L				
1	L16B	L64B	LU16B	LU64B				
2	L16AL	L64AL	LU16AL	LU64AL				
3	L16AB	L64AB	LU16AB	LU64AB				
4	L32L	L128L	LU32L	L8				
5	L32B	L128B	LU32B	LU8				
6	L32AL	L128AL	LU32AL					
7	L32AB	L128AB	LU32AB					

minor operation code field values for L.MINOR

S.MINOR	0	8	16	24	32	40	48	56
0	S16L	S64L	SAS64AL					
1	S16B	S64B	SAS64AB					
2	S16AL	S64AL	SCS64AL	SDCS64AL				
3	S16AB	S64AB	SCS64AB	SDCS64AB				
4	S32L	S128L	SMS64AL	SB				
5	S32B	S128B	SMS64AB					
6	S32AL	S128AL	SMUS64AL					
7	S32AB	S128AB	SMUS64AB					

minor operation code field values for S.MINOR

G.size	0	8	16	24	32	40	48	56
0			GSETE	GSETEF	GADDHN	GSUBHN	GMULADD	GADDL
1	GADD		GSETNE	GSETLGF	GADDHZ	GSUBHZ		GADDLU
2	GADDO		GSETANDE	GSETLF	GADDHF	GSUBHF		GAAA
3	GADDUO		GSETANDNE	GSETGEF	GADDHC	GSUBHC		
4			GSETL.Z	GSETEF.X	GADDHUN	GSUBHUN	GMULSUB	GSUBL
5	GSUB		GSETGE.GEZ	GSETLGF.X	GADDHUZ	GSUBHUZ		GSUBLU
6	GSUBO		GSETLU.GZ	GSETLF.X	GADDHUF	GSUBHUF		GASA
7	GSUBUO		GSETGEU.LEZ	GSETGEF.X	GADDHUC	GSUBHUC		GCOM

minor operation code field values for G.size

XSHIFTI	0	8	16	24	32	40	48	56
0	XSHLI	XSHLIO		XSHRI		XEXPANDI		XCOMPRESSI
1								
2								
3								
4	XSHLMI	XSHLIOU	XSHRMI	XSHRIU	XROTLI	XEXPANDIU	XROTRI	XCOMPRESSIU
5								
6								
7								

minor operation code field values for XSHIFTI

XSHIFT	0	8	16	24	32	40	48	56
0	XSHL	XSHLO		XSHR		XEXPAND		XCOMPRESS
1								
2								
3								
4	XSHLM	XSHLOU	XSHRM	XSHRU	XROTL	XEXPANDU	XROTR	XCOMPRESSU
5								
6								
7								

minor operation code field values for XSHIFT

E.size	0	8	16	24	32	40	48	56
0	EMULFN	EMULADDFN	EADDFN	ESUBFN	EMUL	EMULADD	EDIVFN	ECON
1	EMULFZ	EMULADDFZ	EADDFZ	ESUBFZ	EMULU	EMULADDU	EDIVFZ	ECONU
2	EMULFF	EMULADDFF	EADDFF	ESUBFF	EMULM	EMULADDM	EDIVFF	ECONM
3	EMULFC	EMULADDFC	EADDFC	ESUBFC	EMULC	EMULADDC	EDIVFC	ECONC
4	EMULFX	EMULADDFX	EADDFX	ESUBFX	EMULSUM	EMULSUB	EDIVFX	EDIV
5	EMULF	EMULADDF	EADDF	ESUBF	EMULSUMU	EMULSUBU	EDIVF	EDIVU
6	EMULCF	EMULADDCF	ECONFL	ECONCFL	EMULSUMM	EMULSUBM	EMULSUMF	EMULP
7	EMULSUMCF	EMULSUBCF	ECONFB	ECONCFB	EMULSUMC	EMULSUBC	EMULSUBF	EUNARY

minor opc.ation code field values for E.size

[illegible]	0	8	16	24	32	40	48	56
0	[illegible]	[illegible]						
1	[illegible]	[illegible]	[illegible]					
2	[illegible]	[illegible]	[illegible]					
3	[illegible]	[illegible]	[illegible]					
4	[illegible]	[illegible]		[illegible]				
5	[illegible]	[illegible]	[illegible]	[illegible]				
6	[illegible]	[illegible]	[illegible]	[illegible]				
7	[illegible]	[illegible]	[illegible]	[illegible]				

minor operation code field values for W.MINOR.L or W.MINOR.B

For the major operation field values E.MUL.X.I, E.MUL.X.I.U, E.MUL.X.I.M, E.MUL.X.I.C, E.MUL.ADD.X.I, E.MUL.ADD.X.I.U, E.MUL.ADD.X.I.M, E.MUL.ADD.X.I.C, E.CON.X.I.L, E.CON.X.I.B, E.CON.X.I.U.L, E.CON.X.I.U.B, E.CON.X.I.M.L, E.CON.X.I.M.B, E.CON.X.I.C.L, E.CON.X.I.C.B, E.EXTRACT.I, E.EXTRACT.I.U, W.MUL.MAT.X.I.U.L, W.MUL.MAT.X.I.U.B, W.MUL.MAT.X.I.M.L, W.MUL.MAT.X.I.M.B, W.MUL.MAT X.I.C.L, and W.MUL.MAT.X.I.C.B, another six bits in the instruction specify a minor operation code, which indicates operand size, rounding, and shift amount:

31 … 24	23 … 6	5 … 0
major	other	minor
8	18	6

The minor field is filled with a value from the following table: Note that the shift amount field value shown below is the "sh" value, which is encoded in an instruction-dependent manner from the immediate field in the assembler format.

XI	0	8	16	24	32	40	48	56
0	8F.0	8N.0	16F.0	16N.0	32F.0	32N.0	64F.0	64N.0
1	8F.1	8N.1	16F.1	16N.1	32F.1	32N.1	64F.1	64N.1
2	8F.2	8N.2	16F.2	16N.2	32F.2	32N.2	64F.2	64N.2
3	8F.3	8N.3	16F.3	16N.3	32F.3	32N.3	64F.3	64N.3
4	8Z.0	8C.0	16Z.0	16C.0	32Z.0	32C.0	64Z.0	64C.0
5	8Z.1	8C.1	16Z.1	16C.1	32Z.1	32C.1	64Z.1	64C.1
6	8Z.2	8C.2	16Z.2	16C.2	32Z.2	32C.2	64Z.2	64C.2
7	8Z.3	8C.3	16Z.3	16C.3	32Z.3	32C.3	64Z.3	64C.3

minor operation code field values for EMULXI, EMULXIU, EMULXIM, EMULXIC, EMULADDXI, EMULADDXIU, EMULADDXIM, EMULADDXIC, ECONXIL, ECONXIB, ECONXIUL, ECONXIUB, ECONXIML, ECONXIMB, ECONXICL, ECONXICB, EEXTRACTI, EEXTRACTIU, WMULMATXIUL, WMULMATXIUB, WMULMATXIML, WMULMATXIMB, WMULMATXICL, and WMULMATXICB,

For the major operation field values GCOPYI, two bits in the instruction specify an operand size:

31 … 24	23 … 18	17 16	15 … 0
op	rd	sz	imm
8	6	2	16

For the major operation field values G.AND.I, G.NAND.I, G.NOR.I, G.OR.I, G.XOR.I, G.ADD.I, G.ADD.I.O, G.ADD.I.UO, G.SET.AND.E.I, G.SET.AND.NE.I, G.SET.E.I, G.SET.GE.I, G.SET.L.I, G.SET.NE.I, G.SET.GE.I.U, G.SET.L.I.U, G.SUB.I, G.SUB.I.O, G.SUB.I.UO, two bits in the instruction specify an operand size:

31–24	23–18	17–12	11–10	9–0
op	rd	rc	sz	imm
8	6	6	2	10

The sz field is filled with a value from the following table:

sz	size
0	16
1	32
2	64
3	128

operand size field values for G.COPY.I, G.AND.I, G.NAND.I, G.NOR.I, G.OR.I, G.XOR.I, G.ADD.I, G.ADD.I.O, G.ADD.I.UO, G.SET.AND.E.I, G.SET.AND.NE.I, G.SET.E.I, G.SET.GE.I, G.SET.L.I, G.SET.NE.I, G.SET.GE.I.U, G.SET.L.I.U, G.SUB.I, G.SUB.I.O, G.SUB.I.UO

For the major operation field values E.8, E.16, E.32, E.64, E.128, with minor operation field value E.UNARY, another six bits in the instruction specify a unary operation code:

31–24	23–18	17–12	11–6	5–0
major	rd	rc	unary	minor
8	6	6	6	6

The unary field is filled with a value from the following table:

E.UNARY	0	8	16	24	32	40	48	56
0	E.SQR.F.N	E.SUM.F.N	E.SINK.F.N	E.FLOAT.F.N	E.DEFLATE.F.N	E.SUM		
1	E.SQR.F.Z	E.SUM.F.Z	E.SINK.F.Z	E.FLOAT.F.Z	E.DEFLATE.F.Z	E.SUM.U	[illegible]	
2	E.SQR.F.F	E.SUM.F.F	E.SINK.F.F	E.FLOAT.F.F	E.DEFLATE.F.F	E.LOG.MOST	[illegible]	
3	E.SQR.F.C	E.SUM.F.C	E.SINK.F.C	E.FLOAT.F.C	E.DEFLATE.F.C	E.LOG.MOST.U	[illegible]	
4	E.SQR.F.X	E.SUM.F.X	E.SINK.F.X	E.FLOAT.F.X	E.DEFLATE.F.X			
5	E.SQR.F	E.SUM.F	E.SINK.F	E.FLOAT.F	E.DEFLATE.F			
6	E.RSQR.EST.F.X	E.RECEST.F.X	E.ABS.F.X	E.NEG.F.X	E.INFLATE.F.X		E.COPY.F.X	
7	E.RSQR.EST.F	E.RECEST.F	E.ABS.F	E.NEG.F	E.INFLATE.F		E.COPY.F	

unary operation code field values for E.UNARY.size

For the major operation field values A.MINOR and G.MINOR, with minor operation field values A.COM and G.COM, another six bits in the instruction specify a comparison operation code:

31–24	23–18	17–12	11–6	5–0
major	rd	rc	compare	minor
8	6	6	6	6

The compare field is filled with a value from the following table:

xCOM	0	8	16	24	32	40	48	56
0	[illegible]	xCOMEF						
1	[illegible]	xCOMLGF						
2	xCOMAND	xCOMLF						
3	xCOMANDNE	xCOMGEF						
4	xCOML	xCOMEFX						
5	xCOMGE	xCOMLGFX						
6	xCOMLU	xCOMLFX						
7	xCOMGEU	xCOMGEFX						

compare operation code field values for ACOM.op and G.COM.op.size

General Forms

The general forms of the instructions coded by a major operation code are one of the following:

31 … 24	23 … 0
major	offset
8	24

31 … 24	23 … 18	17 … 0
major	rd	offset
8	6	18

31 … 24	23 … 18	17 … 12	11 … 0
major	rd	rc	offset
8	6	6	12

31 … 24	23 … 18	17 … 12	11 … 6	5 … 0
major	rd	rc	rb	ra
8	6	6	6	6

The general forms of the instructions coded by major and minor operation codes are one of the following:

31 … 24	23 … 18	17 … 12	11 … 6	5 … 0
major	rd	rc	rb	minor
8	6	6	6	6

31 … 24	23 … 18	17 … 12	11 … 6	5 … 0
major	rd	rc	simm	minor
8	6	6	6	6

The general form of the instructions coded by major, minor, and unary operation codes is the following:

31 … 24	23 … 18	17 … 12	11 … 6	5 … 0
major	rd	rc	unary	minor
8	6	6	6	6

Register rd is either a source register or destination register, or both. Registers rc and rb are always source registers. Register ra is always a destination register.

Instruction Fetch

Definition

```
def Thread(th) as
    forever do
        catch exception
            if (EventRegister & EventMask[th]) ≠ 0 then
                if ExceptionState=0 then
                    raise EventInterrupt
                endif
            endif
            inst ← LoadMemoryX(ProgramCounter,ProgramCounter,32,L)
            Instruction(inst)
        endcatch
        case exception of
            EventInterrupt,
            ReservedInstruction,
            AccessDisallowedByVirtualAddress,
            AccessDisallowedByTag,
            AccessDisallowedByGlobalTB,
            AccessDisallowedByLocalTB,
            AccessDetailRequiredByTag,
            AccessDetailRequiredByGlobalTB,
            AccessDetailRequiredByLocalTB,
            MissInGlobalTB,
            MissInLocalTB,
            FixedPointArithmetic,
            FloatingPointArithmetic,
            GatewayDisallowed:
                case ExceptionState of
                    0:
                        PerformException(exception)
                    1:
                        PerformException(SecondException)
                    2:
                        PerformMachineCheck(ThirdException)
                endcase
            TakenBranch:
                ContinuationState ← (ExceptionState=0) ? 0 : ContinuationState
            TakenBranchContinue:
                /* nothing */
            none, others:
                ProgramCounter ← ProgramCounter + 4
                ContinuationState ← (ExceptionState=0) ? 0 : ContinuationState
        endcase
    endforever
enddef
```

Perform Exception

Definition

```
def PerformException(exception) as
    v ← (exception > 7) ? 7 : exception
    t ← LoadMemory(ExceptionBase,ExceptionBase+Thread*128+64+8*v,64,L)
    if ExceptionState = 0 then
        u ← RegRead(3,128) || RegRead(2,128) || RegRead(1,128) || RegRead(0,128)
        StoreMemory(ExceptionBase,ExceptionBase+Thread*128,512,L,u)
        RegWrite(0,64,ProgramCounter63..2 || PrivilegeLevel
        RegWrite(1,64,ExceptionBase+Thread*128)
        RegWrite(2,64,exception)
        RegWrite(3,64,FailingAddress)
    endif
    PrivilegeLevel ← t1..0
    ProgramCounter ← t63..2 || 0^2
    case exception of
        AccessDetailRequiredByTag,
        AccessDetailRequiredByGlobalTB,
        AccessDetailRequiredByLocalTB:
            ContinuationState ← ContinuationState + 1
        others:
            /* nothing */
    endcase
    ExceptionState ← ExceptionState + 1
enddef
```

Instruction Decode

```
def instruction(inst) as
    major ← inst31..24
    rd ← inst23..18
    rc ← inst17..12
    simm ← rb ← inst11..6
    minor ← ra ← inst5..0
    case major of
        ARES:
            AlwaysReserved
        AMINOR:
            minor ← inst5..0
            case minor of
                AADD, AADD.O, AADD.OU, AAND, AANDN, ANAND, ANOR,
                AOR, AORN, AXNOR, AXOR:
                    Address(minor,rd,rc,rb)
                ACOM:
                    compare ← inst11..6
                    case compare of
                        ACOM.E, ACOM.NE, ACOM.AND.E, ACOM.AND.NE,
                        ACOM.L, ACOM.GE, ACOM.LU, ACOM.GE.U:
                            AddressCompare(compare,rd,rc)
                        others:
                            raise ReservedInstruction
```

```
                endcase
            ASUB, ASUB.O, ASUB.U.O,
            ASET.AND.E, ASET.AND.NE, ASET.E, ASET.NE,
            ASET.L, ASET.GE, ASET.L.U, ASET.GE.U:
                AddressReversed(minor,rd,rc,rb)
            ASHL.I.ADD..ASHL.I.ADD+3:
                AddressShiftLeftImmediateAdd(inst1..0,rd,rc,rb)
            ASHL.I.SUB..ASHL.I.SUB+3:
                AddressShiftLeftImmediateSubtract(inst1..0,rd,rc,rb)
            ASHL.I, ASHL.I.O, ASHL.I.U.O, ASHR.I, ASHR.I.U, AROTR.I:
                AddressShiftImmediate(minor,rd,rc,simm)
            others:
                raise ReservedInstruction
        endcase
    ACOPY.I:
        AddressCopyImmediate(major,rd,inst17..0)
    AADD.I, AADD.I.O, AADD.I.U.O, AAND.I, AOR.I, ANAND.I, ANOR.I, AXOR.I:
        AddressImmediate(major,rd,rc,inst11..0)
    ASET.AND.E.I, ASET.AND.NE.I, ASET.E.I, ASET.NE.I,
    ASET.L.I, ESET.GE.I, ASET.L.U.I, ASET.GE.U.I,
    ASUB.I, ASUB.I.O, ASUB.I.U.O:
        AddressImmediateReversed(major,rd,rc,inst11..0)
    AMUX:
        AddressTernary(major,rd,rc,rb,ra)
    B.MINOR:
        case minor of
            B:
                Branch(rd,rc,rb)
            B.BACK:
                BranchBack(rd,rc,rb)
            B.BARRIER:
                BranchBarrier(rd,rc,rb)
            B.DOWN:
                BranchDown(rd,rc,rb)
            B.GATE:
                BranchGateway(rd,rc,rb)
            B.HALT:
                BranchHalt(rd,rc,rb)
            B.HINT:
                BranchHint(rd,inst17..12,simm)
            B.LINK:
                BranchLink(rd,rc,rb)
            others:
                raise ReservedInstruction
        endcase
    BE, BNE, BL, BGE, BLU, BGE.U, BAND.E, BAND.NE:
        BranchConditional(major,rd,rc,inst11..0)
    BHINTI:
        BranchHintImmediate(inst23..18,inst17..12,inst11..0)
    BI:
        BranchImmediate(inst23..0)
    BLINKI:
        BranchImmediateLink(inst23..0)
    BEF16, BLGF16, BLF16, BGEF16,
    BEF32, BLGF32, BLF32, BGEF32,
    BEF64, BLGF64, BLF64, BGEF64,
    BEF128, BLGF128, BLF128, BGEF128:
```

```
        BranchConditionalFloatingPoint(major,rd,rc,inst11..0)
    BIF32, BNIF32, BNVF32, BVF32:
        BranchConditionalVisibilityFloatingPoint(major,rd,rc,inst11..0)
    LMINOR
        case minor of
            L16L, LU16L, L32L, LU32L, L64L, LU64L, L128L, L8, LU8,
            L16AL, LU16AL, L32AL, LU32AL, L64AL, LU64AL, L128AL,
            L16B, LU16B, L32B, LU32B, L64B, LU64B, L128B,
            L16AB, LU16AB, L32AB, LU32AB, L64AB, LU64AB, L128AB:
                Load(minor,rd,rc,rb)
            others:
                raise ReservedInstruction
        endcase
    LI16L, LIU16L, LI32L, LIU32L, LI64L, LIU64L, LI128L, LI8, LIU8,
    LI16AL, LIU16AL, LI32AL, LIU32AL, LI64AL, LIU64AL, LI128AL,
    LI16B, LIU16B, LI32B, LIU32B, LI64B, LIU64B, LI128B,
    LI16AB, LIU16AB, LI32AB, LIU32AB, LI64AB, LIU64AB, LI128AB:
        LoadImmediate(major,rd,rc,inst11..0)
    SMINOR
        case minor of
            S16L, S32L, S64L, S128L, S8,
            S16AL, S32AL, S64AL, S128AL,
            SAS64AL, SCS64AL, SMS64AL, SM64AL,
            S16B, S32B, S64B, S128B,
            S16AB, S32AB, S64AB, S128AB,
            SAS64AB, SCS64AB, SMS64AB, SM64AB:
                Store(minor,rd,rc,rb)
            SDCS64AB, SDCS64AL:
                StoreDoubleCompareSwap(minor,rd,rc,rb)
            others:
                raise ReservedInstruction
        endcase
    SI16L, SI32L, SI64L, SI128L, SI8,
    SI16AL, SI32AL, SI64AL, SI128AL,
    SASI64AL, SCSI64AL, SMSI64AL, SMUXI64AL,
    SI16B, SI32B, SI64B, SI128B,
    SI16AB, SI32AB, SI64AB, SI128AB
    SASI64AB, SCSI64AB, SMSI64AB, SMUXI64AB:
        StoreImmediate(major,rd,rc,inst11..0)
    G.8, G.16, G.32, G.64, G.128:
        minor ← inst5..0
        size ← 0 || 1 || 0^(3+major-G.8)
        case minor of
            G.ADD, G.ADD.L, G.ADD.LU, G.ADD.O, G.ADD.OU:
                Group(minor,size,rd,rc,rb)
            G.ADDHC, G.ADDHF, G.ADDHN, G.ADDHZ,
            G.ADDHUC, G.ADDHUF, G.ADDHUN, G.ADDHUZ:
                GroupAddHalve(minor,inst1..0,size,rd,rc,rb)
            G.AAA, G.ASA:
                GroupInplace(minor,size,rd,rc,rb)
            G.SET.AND.E, G.SET.AND.NE, G.SET.E, G.SET.NE,
            G.SET.L, G.SET.GE, G.SET.L.U, G.SET.GE.U:
            G.SUB, G.SUB.L, G.SUB.LU, G.SUB.O, G.SUB.U.O
                GroupReversed(minor,size,ra,rb,rc)
            G.SET.E.F, G.SET.LG.F, G.SET.GE.F, G.SET.L.F,
            G.SET.E.F.X, G.SET.LG.F.X, G.SET.GE.F.X, G.SET.L.F.X:
                GroupReversedFloatingPoint(minor.op, size,
```

```
                        minor.round, rd, rc, rb)
                G.SHL.I.ADD..G.SHL.I.ADD+3:
                        GroupShiftLeftImmediateAdd(inst1..0,size,rd,rc,rb)
                G.SHL.I.SUB..G.SHL.I.SUB+3:
                        GroupShiftLeftImmediateSubtract(inst1..0,size,rd,rc,rb)
                G.SUBHC, G.SUBHF, G.SUBHN, G.SUBHZ,
                G.SUBHUC, G.SUBHUF, G.SUBHUN, G.SUBHUZ:
                        GroupSubtractHalve(minor,inst1..0,size,rd,rc,rb)
                G.COM:
                        compare ← inst11..6
                        case compare of
                                G.COM.E, G.COM.NE, G.COM.AND.E, G.COM.AND.NE,
                                G.COM.L, G.COM.GE, G.COM.L.U, G.COM.GE.U:
                                        GroupCompare(compare,size,ra,rb)
                                others:
                                        raise ReservedInstruction
                        endcase
                others:
                        raise ReservedInstruction
        endcase
G.BOOLEAN..G.BOOLEAN+1:
        GroupBoolean(major,rd,rc,rb,minor)
G.COPY.I...G.COPY.I+1:
        size ← 0 || 1 || 0^(4+inst17..16)
        GroupCopyImmediate(major,size,rd,inst15..0)
G.AND.I, G.NAND.I, G.NOR.I, G.OR.I, G.XOR.I,
G.ADD.I, G.ADD.I.O, G.ADD.I.U.O:
        size ← 0 || 1 || 0^(4+inst11..10)
        GroupImmediate(major,size,rd,rc,inst9..0)
G.SET.AND.E.I, G.SET.AND.NE.I, G.SET.E.I, G.SET.GE.I, G.SET.L.I,
G.SET.NE.I, G.SET.GE.I.U, G.SET.L.I.U, G.SUB.I, G.SUB.I.O, G.SUB.I.U.O:
        size ← 0 || 1 || 0^(4+inst11..10)
        GroupImmediateReversed(major,size,rd,rc,inst9..0)
G.MUX:
        GroupTernary(major,rd,rc,rb,ra)
X.SHIFT:
        minor ← inst5..2 || 0^2
        size ← 0 || 1 || 0^(inst24 || inst1..0)
        case minor of
                X.EXPAND, X.UEXPAND, X.SHL, X.SHL.O, X.SHL.U.O,
                X.ROTR, X.SHR, X.SHR.U,
                        Crossbar(minor,size,rd,rc,rb)
                X.SHL.M, X.SHR.M:
                        CrossbarInplace(minor,size,rd,rc,rb)
                others:
                        raise ReservedInstruction
        endcase
X.EXTRACT:
        CrossbarExtract(major,rd,rc,rb,ra)
X.DEPOSIT, X.DEPOSIT.U, X.WITHDRAW, X.WITHDRAW.U:
        CrossbarField(major,rd,rc,inst11..6,inst5..0)
X.DEPOSIT.M:
        CrossbarFieldInplace(major,rd,rc,inst11..6,inst5..0)
X.SHIFT.I:
        minor ← inst5..0
```

```
        case minor5..2 || 0^2 of
            X.COMPRESS.I, X.EXPAND.I, X.ROTR.I, X.SHL.I, X.SHL.I.O, X.SHL.I.U.O,
            X.SHR.I, X.COMPRESS.I.U, X.EXPAND.I.U, X.SHR.U.I:
                CrossbarShortImmediate(minor,rd,rc,simm)
            X.SHL.M.I, X.SHR.M.I:
                CrossbarShortImmediateInplace(minor,rd,rc,simm)
            others
                raise ReservedInstruction
        endcase
    X.SHUFFLE, X.SHUFFLE+1
        CrossbarShuffle(major,rd,rc,rb,simm)
    X.SWIZZLE, X.SWIZZLE+3
        CrossbarSwizzle(major,rd,rc, inst11..6,inst5..0)
    X.SELECT.8:
        CrossbarTernary(major,rd,rc,rb,ra)
    E.8, E.16, E.32, E.64, E.128:
        minor ← inst5..0
        size ← 0 || 1 || 0^(3+major-E.8)
        case minor of
            E.CON, E.CON.U, E.CON.M, E.CON.C,
            E.MUL, E.MUL.U, E.MUL.M, E.MUL.C,
            E.MUL.SUM, E.MUL.SUM.U, E.MUL.SUM.M, E.MUL.SUM.C,
            E.DIV, E.DIV.U, E.MUL.P:
                Ensemble(minor,size,ra,rb,rc)
            E.CON.F.L, E.CON.F.B, E.CON.C.F.L, E.CON.C.F.B:
                EnsembleConvolveFloatingPoint(minor,size,rd,rc,rb)
            E.ADD.F.N, E.MUL.C.F.N, E.MUL.F.N, E.DIV.F.N,
            E.ADD.F.Z, E.MUL.C.F.Z, E.MUL.F.Z, E.DIV.F.Z,
            E.ADD.F.F, E.MUL.C.F.F, E.MUL.F.F, E.DIV.F.F,
            E.ADD.F.C, E.MUL.C.F.C, E.MUL.F.C, E.DIV.F.C,
            E.ADD.F, E.MUL.C.F, E.MUL.F, E.DIV.F,
            E.ADD.F.X, E.MUL.C.F.X, E.MUL.F.X, E.DIV.F.X:
                EnsembleFloatingPoint(minor.op, major.size, minor.round, rd, rc, rb)
            E.MUL.ADD, E.MUL.ADD.U, E.MUL.ADD.M, E.MUL.ADD.C:
                EnsembleInplace(minor,size,rd,rc,rb)
            E.MUL.SUB, E.MUL.SUB.U, E.MUL.SUB.M, E.MUL.SUB.C:
                EnsembleInplaceReversed(minor,size,rd,rc,rb)
            E.MUL.SUB.F, E.MUL.SUB.C.F:
                EnsembleInplaceReversedFloatingPoint(minor,size,rd,rc,rb)
            E.SUB.F.N, E.SUB.F.Z, E.SUB.F.F, E.SUB.F.C, E.SUB.F, E.SUB.F.X:
                EnsembleReversedFloatingPoint(minor.op, major.size,
                    minor.round, rd, rc, rb)
            E.UNARY:
                case unary of
                    E.SUM, E.SUM.U, E.LOG.MOST, E.LOG.MOST.U:
                        EnsembleUnary(unary,rd,rc)
                    E.ABS.F, E.ABS.F.X, E.COPY.F, E.COPY.F.X,
                    E.DEFLATE.F, E.DEFLATE.F.N, E.DEFLATE.F.Z,
                    E.DEFLATE.F.F, E.DEFLATE.F.C, E.DEFLATE.F.X:
                    E.FLOAT.F, E.FLOAT.F.N, E.FLOAT.F.Z,
                    E.FLOAT.F.F, E.FLOAT.F.C, E.FLOAT.F.X:
                    E.INFLATE.F, E.INFLATE.F.X, E.NEG.F, E.NEG.F.X,
                    E.RECEST.F, E.RECEST.F.X, E.RSQREST.F, E.RSQREST.F.X,
                    E.SQR.F, E.SQR.F.N, E.SQR.F.Z, E.SQR.F.F, E.SQR.F.C, E.SQR.F.X,
                    E.SUM.F, E.SUM.F.N, E.SUM.F.Z,
                    E.SUM.F.F, E.SUM.F.C, E.SUM.F.X:
                    E.SINK.F, E.SINK.F.Z.D, E.SINK.F.F.D, E.SINK.F.C.D, E.SINK.F.X.D,
                    E.SINK.F.N, E.SINK.F.Z, E.SINK.F.F, E.SINK.F.C, E.SINK.F.X:
```

```
                                EnsembleUnaryFloatingPoint(unary.op, major.size,
                                        unary.round, rd, rc)
                        others:
                                raise ReservedInstruction
                endcase
        others:
                raise ReservedInstruction
    endcase
E.CON.X.I.L, E.CON.X.I.B, E.CON.X.I.U.L, E.CON.X.I.U.B,
E.CON.X.I.M.L, E.CON.X.I.M.B, E.CON.X.I.C.L, E.CON.X.I.C.B
    size ← 1 || 0^(3+inst5..4)
    EnsembleConvolveExtractImmediate(major,inst3..2,size,rd,rc,rb,inst1..0)
E.MUL.X, E.EXTRACT, E.SCAL.ADD.X:
    EnsembleExtract(major,rd,rc,rb,ra)
E.EXTRACT.I, E.EXTRACT.I.U, E.MUL.X.I, E.MUL.X.I.U, E.MUL.X.I.M, E.MUL.X.I.C:
    size ← 1 || 0^(3+inst5..4)
    EnsembleExtractImmediate(major,inst3..2,size,rd,rc,rb,inst1..0)
E.MUL.ADD.X.I, E.MUL.ADD.X.I.U, E.MUL.ADD.X.I.M, E.MUL.ADD.X.I.C:
    size ← 1 || 0^(3+inst5..4)
    EnsembleExtractImmediateInplace(major,inst3..2,size,rd,rc,rb,inst1..0)
E.MUL.GAL.8, E.MUL.GAL.64:
    size ← 1 || 0^(3+inst26..24)
    EnsembleTernary(major,size,rd,rc,rb,ra)
E.MUL.ADD.F16, E.MUL.ADD.F32, E.MUL.ADD.F64, E.MUL.ADD.F128
E.MUL.SUB.F16, E.MUL.SUB.F32, E.MUL.SUB.F64, E.MUL.SUB.F128,
E.SCAL.ADD.F16, E.SCAL.ADD.F32, E.SCAL.ADD.F64:
    EnsembleTernaryFloatingPoint(major,rd,rc,rb,ra)
W.MINOR.B, W.MINOR.L:
    case minor of
        W.TRANSLATE.8, W.TRANSLATE.16, W.TRANSLATE.32, W.TRANSLATE.64:
            size ← 1 || 0^(3+inst5..4)
            WideTranslate(major,size,rd,rc,rb)
        W.MUL.MAT.8, W.MUL.MAT.16, W.MUL.MAT.32, W.MUL.MAT.64,
        W.MUL.MAT.U.8, W.MUL.MAT.U.16, W.MUL.MAT.U.32, W.MUL.MAT.U.64,
        W.MUL.MAT.M.8, W.MUL.MAT.M.16, W.MUL.MAT.M.32, W.MUL.MAT.M.64,
        W.MUL.MAT.C.8, W.MUL.MAT.C.16, W.MUL.MAT.C.32, W.MUL.MAT.C.64,
        W.MUL.MAT.P.8, W.MUL.MAT.P.16, W.MUL.MAT.P.32, W.MUL.MAT.P.64:
            size ← 1 || 0^(3+inst5..4)
            WideMultiply(major,minor,size,rd,rc,rb)
        W.MUL.MAT.F16, W.MUL.MAT.F.32, W.MUL.MAT.F64,
        W.MUL.MAT.C.F16, W.MUL.MAT.C.F32, W.MUL.MAT.C.F64:
            size ← 1 || 0^(3+inst5..4)
            WideFloatingPointMultiply(major,minor,size,rd,rc,rb)
        others:
    endcase
W.MUL.MAT.X.B, W.MUL.MAT.X.L:
    WideExtract(major,ra,rb,rc,rd)
W.MUL.MAT.X.I.B, W.MUL.MAT.X.I.L, W.MUL.MAT.X.I.U.B, W.MUL.MAT.X.I.U.L,
W.MUL.MAT.X.I.M.B, W.MUL.MAT.X.I.M.L, W.MUL.MAT.X.I.C.B, W.MUL.MAT.X.I.C.L:
    size ← 1 || 0^(3+inst5..4)
    WideExtractImmediate(major,inst3..2,size,ra,rb,rc,inst1..0)
W.MUL.MAT.G.B, W.MUL.MAT.G.L:
    WideMultiplyGalois(major,rd,rc,rb,ra)
W.SWITCH.B, W.SWITCH.L:
    WideSwitch(major,rd,rc,rb,ra)
others:
```

```
            raise ReservedInstruction
    endcase
enddef
```

Always Reserved

This operation generates a reserved instruction exception.

Operation code

A.RES	Always reserved

Format

A.RES imm

ares(imm)

31 … 24	23 … 0
A.RES	imm
8	24

Description

The reserved instruction exception is raised. Software may depend upon this major operation code raising the reserved instruction exception in all implementations. The choice of operation code intentionally ensures that a branch to a zeroed memory area will raise an exception.

Definition

```
def AlwaysReserved as
    raise ReservedInstruction
enddef
```

Exceptions

Reserved Instruction

Address

These operations perform calculations with two general register values, placing the result in a general register.

Operation codes

AADD	Address add
AADD.O	Address add signed check overflow
AADD.U.O	Address add unsigned check overflow
AAND	Address and
AANDN	Address and not
ANAND	Address not and
ANOR	Address not or
AOR	Address or
AORN	Address or not
AXNOR	Address exclusive nor
AXOR	Address xor

Redundancies

AOR rd=rc,rc	⇔	*ACOPY rd=rc*
AAND rd=rc,rc	⇔	*ACOPY rd=rc*
ANAND rd=rc,rc	⇔	*ANOT rd=rc*
ANOR rd=rc,rc	⇔	*ANOT rd=rc*
AXNOR rd=rc,rc	⇔	*ASET rd*
AXOR rd=rc,rc	⇔	*AZERO rd*
AADD rd=rc,rc	⇔	ASHL.I rd=rc,1
AADD.O rd=rc,rc	⇔	ASHL.I.O rd=rc,1
AADD.U.O rd=rc,rc	⇔	ASHL.I.U.O rd=rc,1

Selection

class	operation				check		
arithmetic	ADD				NONE	O	U.O
bitwise	OR	AND	XOR	ANDN			
	NOR	NAND	XNOR	ORN			

Format

op rd=rc,rb

rd=op(rc,rb)

31 — 24	23 — 18	17 — 12	11 — 6	5 — 0
A.MINOR	rd	rc	rb	op
8	6	6	6	6

Description

The contents of registers rc and rb are fetched and the specified operation is performed on these operands. The result is placed into register rd.

Definition

```
def Address(op,rd,rc,rb) as
    c ← RegRead(rc, 64)
    b ← RegRead(rb, 64)
    case op of
        AADD:
            a ← c + b
        AADD.O:
            t ← (c63 || c) + (b63 || b)
            if t64 ≠ t63 then
                raise FixedPointArithmetic
            endif
            a ← t63..0
        AADD.UO:
            t ← (0^1 || c) + (0^1 || b)
            if t64 ≠ 0 then
                raise FixedPointArithmetic
            endif
            a ← t63..0
        AAND:
            a ← c and b
        AOR:
            a ← c or b
        AXOR:
            a ← c xor b:
        AANDN:
            a ← c and not b
        ANAND:
            a ← not (c and b)
        ANOR:
            a ← not (c or b)
        AXNOR:
            a ← not (c xor b)
        AORN:
            a ← c or not b
    endcase
```

```
    RegWrite(rd, 64, a)
enddef
```

Exceptions

Fixed-point arithmetic

Address Compare

These operations perform calculations with two general register values and generate a fixed-point arithmetic exception if the condition specified is met.

Operation codes

ACOM.AND.E	Address compare and equal zero
ACOM.AND.NE	Address compare and not equal zero
ACOM.E	Address compare equal
ACOM.GE	Address compare greater equal signed
ACOM.GE.U	Address compare greater equal unsigned
ACOM.L	Address compare less signed
ACOM.L.U	Address compare less unsigned
ACOM.NE	Address compare not equal

Equivalencies

ACOM.E.Z	Address compare equal zero
ACOM.G.Z	Address compare greater zero signed
ACOM.GE.Z	Address compare greater equal zero signed
ACOM.L.Z	Address compare less zero signed
ACOM.LE.Z	Address compare less equal zero signed
ACOM.NE.Z	Address compare not equal zero
ACOM.G	Address compare greater signed
ACOM.G.U	Address compare greater unsigned
ACOM.LE	Address compare less equal signed
ACOM.LE.U	Address compare less equal unsigned
AFIX	Address fixed point arithmetic exception
ANOP	Address no operation

ACOM.E.Z rc	←	ACOM.AND.E rc,rc
ACOM.G.Z rc	⇐	ACOM.L.U rc,rc
ACOM.GE.Z rc	⇐	ACOM.GE rc,rc
ACOM.L.Z rc	⇐	ACOM.L rc,rc
ACOM.LE.Z rc	⇐	ACOM.GE.U rc,rc
ACOM.NE.Z rc	←	ACOM.AND.NE rc,rc
ACOM.G rc,rd	→	ACOM.L rd,rc
ACOM.G.U rc,rd	→	ACOM.L.U rd,rc
ACOM.LE rc,rd	→	ACOM.GE rd,rc
ACOM.LE.U rc,rd	→	ACOM.GE.U rd,rc
AFIX	←	ACOM.E 0,0
ANOP	←	ACOM.NE 0,0

Redundancies

ACOM.E rd,rd	⇔	*AFIX*
ACOM.NE rd,rd	⇔	*ANOP*

Selection

class	operation	cond	operand
boolean	COMAND COM	E NE	
arithmetic	COM	L GE G LE	NONE U
	COM	*L GE G LE E NE*	*Z*

Format

ACOM.op rd,rc

acomop(rd,rc)
acomopz(rcd)

31–24	23–18	17–12	11–6	5–0
A.MINOR	rd	rc	op	ACOM
8	6	6	6	6

Description

The contents of registers rd and rc are fetched and the specified condition is calculated on these operands. If the specified condition is true, a fixed-point arithmetic exception is generated. This instruction generates no general register results.

Definition

```
def AddressCompare(op,rd,rc) as
    d ← RegRead(rd, 128)
    c ← RegRead(rc, 128)
    case op of
        ACOM.E:
            a ← d = c
        ACOM.NE:
            a ← d ≠ c
        ACOM.AND.E:
            a ← (d and c) = 0
        ACOM.AND.NE:
            a ← (d and c) ≠ 0
        ACOM.L:
            a ← (rd = rc) ? (c < 0) : (d < c)
        ACOM.GE:
            a ← (rd = rc) ? (c ≥ 0) : (d ≥ c)
        ACOM.L.U:
            a ← (rd = rc) ? (c > 0) : ((0 || d) < (0 || c))
        ACOM.GE.U:
            a ← (rd = rc) ? (c ≤ 0) : ((0 || d) ≥ (0 || c))
    endcase
```

```
    if a then
        raise FixedPointArithmetic
    endif
enddef
```

Exceptions

Fixed-point arithmetic

Address Copy Immediate

This operation produces one immediate value, placing the result in a general register.

Operation codes

ACOPY.I	Address copy immediate

Equivalencies

ASET	Address set
AZERO	Address zero

ASET rd	← ACOPY.I rd=-1
AZERO rd	← ACOPY.I rd=0

Format

ACOPY.I rd=imm

rd=acopyi(imm)

31 24	23 18	17 0
A.COPY.I	rd	imm
8	6	18

Description

An immediate value is sign-extended from the 18-bit imm field. The result is placed into register rd.

Definition

```
def AddressCopyImmediate(op,rd,imm) as
    a ← (imm17)110 || imm
    RegWrite(rd, 128, a)
enddef
```

Exceptions

none

Address Immediate

These operations perform calculations with one general register value and one immediate value, placing the result in a general register.

Operation codes

AADD.I	Address add immediate
AADD.I.O	Address add immediate signed check overflow
AADD.I.U.O	Address add immediate unsigned check overflow
AAND.I	Address and immediate
ANAND.I	Address not and immediate
ANOR.I	Address not or immediate
AOR.I	Address or immediate
AXOR.I	Address xor immediate

Equivalencies

AANDN.I	Address and not immediate
ACOPY	Address copy
ANOT	Address not
AORN.I	Address or not immediate
AXNOR.I	Address xnor immediate

AANDN.I rd=rc,imm	→	AAND.I rd=rc,-imm
ACOPY rd=rc	←	AOR.I rd=rc,0
ANOT rd=rc	←	ANOR.I rd=rc,0
AORN.I rd=rc,imm	→	AOR.I rd=rc,-imm
AXNOR.I rd=rc,imm	→	AXOR.I rd=rc,-imm

Redundancies

AADD.I rd=rc,0	⇔	*ACOPY rd=rc*
AADD.I.O rd=rc,0	⇔	*ACOPY rd=rc*
AADD.I.U.O rd=rc,0	⇔	*ACOPY rd=rc*
AAND.I rd=rc,0	⇔	*AZERO rd*
AAND.I rd=rc,-1	⇔	*ACOPY rd=rc*
ANAND.I rd=rc,0	⇔	*ASET rd*
ANAND.I rd=rc,-1	⇔	*ANOT rd=rc*
AOR.I rd=rc,-1	⇔	*ASET rd*
ANOR.I rd=rc,-1	⇔	*AZERO rd*
AXOR.I rd=rc,0	⇔	*ACOPY rd=rc*
AXOR.I rd=rc,-1	⇔	*ANOT rd=rc*

Selection

class	operation	check
arithmetic	ADD	NONE O UO
bitwise	AND OR NAND NOR XOR	

Format

op rd=rc,imm

rd=op(rc,imm)

31 24	23 18	17 12	11 0
op	rd	rc	imm
8	6	6	12

Description

The contents of register rc is fetched, and a 64-bit immediate value is sign-extended from the 12-bit imm field. The specified operation is performed on these operands. The result is placed into register rd.

Definition

```
def AddressImmediate(op,rd,rc,imm) as
    i ← imm_11^52 || imm
    c ← RegRead(rc, 64)
    case op of
        AAND.I:
            a ← c and i
        AOR.I:
            a ← c or i
        ANAND.I:
            a ← c nand i
        ANOR.I:
            a ← c nor i
        AXOR.I:
            a ← c xor i
        AADD.I:
            a ← c + i
        AADD.I.O:
            t ← (c_63 || c) + (i_63 || i)
            if t_64 ≠ t_63 then
                raise FixedPointArithmetic
            endif
            a ← t_63..0
        AADD.I.U.O:
            t ← (c_63 || c) + (i_63 || i)
            if t_64 ≠ 0 then
                raise FixedPointArithmetic
            endif
            a ← t_63..0
```

```
    endcase
    RegWrite(rd, 64, a)
enddef
```

Exceptions

Fixed-point arithmetic

Address Immediate Reversed

These operations perform calculations with one general register value and one immediate value, placing the result in a general register.

Operation codes

ASET.AND.E.I	Address set and equal immediate
ASET.AND.NE.I	Address set and not equal immediate
ASET.E.I	Address set equal immediate
ASET.GE.I	Address set greater equal immediate signed
ASET.L.I	Address set less immediate signed
ASET.NE.I	Address set not equal immediate
ASET.GE.I.U	Address set greater equal immediate unsigned
ASET.L.I.U	Address set less immediate unsigned
ASUB.I	Address subtract immediate
ASUB.I.O	Address subtract immediate signed check overflow
ASUB.I.U.O	Address subtract immediate unsigned check overflow

Equivalencies

ANEG	Address negate
ANEG.O	Address negate signed check overflow
ASET.G.I.U	Address set greater immediate unsigned
ASET.LE.I	Address set less equal immediate signed
ASET.LE.I.U	Address set less equal immediate unsigned

ANEG rd=rc	→	ASUB.I rd=0,rc
ANEG.O rd=rc	→	ASUB.I.O rd=0,rc
ASET.G.I rd=imm,rc	→	ASET.GE.I rd=imm+1,rc
ASET.G.I.U rd=imm,rc	→	ASET.GE.I.U rd=imm+1,rc
ASET.LE.I rd=imm,rc	→	ASET.L.I rd=imm-1,rc
ASET.LE.I.U rd=imm,rc	→	ASET.L.I.U rd=imm-1,rc

Redundancies

ASETAND.E.I rd=rc,0	⇔	*ASET rd*
ASETAND.NE.I rd=rc,0	⇔	*AZERO rd*
ASETAND.E.I rd=rc,-1	⇔	*ASET.E.Z rd=rc*
ASETAND.NE.I rd=rc,-1	⇔	*ASET.NE.Z rd=rc*
ASET.E.I rd=rc,0	⇔	*ASET.E.Z rd=rc*
ASET.GE.I rd=rc,0	⇔	*ASET.GE.Z rd=rc*
ASET.L.I rd=rc,0	⇔	*ASET.L.Z rd=rc*
ASET.NE.I rd=rc,0	⇔	*ASET.NE.Z rd=rc*
ASET.GE.I.U rd=rc,0	⇔	*ASET.GE.U.Z rd=rc*
ASET.L.I.U rd=rc,0	⇔	*ASET.L.U.Z rd=rc*

Selection

class	operation	cond	form	type	check
arithmetic	SUB		I		
				NONE U	O
boolean	SETAND SET	E NE	I		
	SET	L GE G LE	I	NONE U	

Format

op rd=imm,rc

rd=op(imm,rc)

31 … 24	23 … 18	17 … 12	11 … 0
op	rd	rc	imm
8	6	6	12

Description

The contents of register rc is fetched, and a 64-bit immediate value is sign-extended from the 12-bit imm field. The specified operation is performed on these operands. The result is placed into register rd.

Definition

```
def AddressImmediate(op,rd,rc,imm) as
    i ← imm_11^52 || imm
    c ← RegRead(rc, 64)
    case op of
        ASUB.I:
            a ← i - c
        ASUB.I.O:
            t ← (i_63 || i) - (c_63 || c)
            if t_64 ≠ t_63 then
                raise FixedPointArithmetic
            endif
```

```
            a ← t6..0
        ASUBI.U.O:
            t ← (i63 || i) - (c63 || c)
            if t64 ≠ 0 then
                raise FixedPointArithmetic
            endif
            a ← t63..0
        ASET.AND.E.I:
            a ← ((i and c) = 0)^64
        ASET.AND.NE.I:
            a ← ((i and c) ≠ 0)^64
        ASET.E.I:
            a ← (i = c)^64
        ASET.NE.I:
            a ← (i ≠ c)^64
        ASET.L.I:
            a ← (i < c)^64
        ASET.GE.I:
            a ← (i ≥ c)^64
        ASET.L.I.U:
            a ← ((0 || i) < (0 || c))^64
        ASET.GE.I.U:
            a ← ((0 || i) ≥ (0 || c))^64
    endcase
    RegWrite(rd, 64, a)
enddef
```

Exceptions

Fixed-point arithmetic

Address Reversed

These operations perform calculations with two general register values, placing the result in a general register.

Operation codes

ASET.AND.E	Address set and equal zero
ASET.AND.NE	Address set and not equal zero
ASET.E	Address set equal
ASET.GE	Address set greater equal signed
ASET.GE.U	Address set greater equal unsigned
ASET.L	Address set less signed
ASET.L.U	Address set less unsigned
ASET.NE	Address set not equal
ASUB	Address subtract
ASUB.O	Address subtract signed check overflow
ASUB.U.O	Address subtract unsigned check overflow

Equivalencies

ASET.E.Z	Address set equal zero
ASET.G.Z	Address set greater zero signed
ASET.GE.Z	Address set greater equal zero signed
ASET.L.Z	Address set less zero signed
ASET.LE.Z	Address set less equal zero signed
ASET.NE.Z	Address set not equal zero
ASET.G	Address set greater signed
ASET.G.U	Address set greater unsigned
ASET.LE	Address set less equal signed
ASET.LE.U	Address set less equal unsigned

ASET.E.Z rd=rc	←	ASET.AND.E rd=rc,rc
ASET.G.Z rd=rc	⇐	ASET.L.U rd=rc,rc
ASET.GE.Z rd=rc	⇐	ASET.GE rd=rc,rc
ASET.L.Z rd=rc	⇐	ASET.L rd=rc,rc
ASET.LE.Z rd=rc	⇐	ASET.GE.U rd=rc,rc
ASET.NE.Z rd=rc	←	ASET.AND.NE rd=rc,rc
ASET.G rd=rb,rc	→	ASET.L rd=rc,rb
ASET.G.U rd=rb,rc	→	ASET.L.U rd=rc,rb
ASET.LE rd=rb,rc	→	ASET.GE rd=rc,rb
ASET.LE.U rd=rb,rc	→	ASET.GE.U rd=rc,rb

Redundancies

ASET.E rd=rc,rc	⇔	*ASET rd*
ASET.NE rd=rc,rc	⇔	*AZERO rd*

Selection

class	operation	cond	operand	check
arithmetic	SUB			
			NONE U	O
boolean	SET.AND SET	E NE		
	SET	L GE G LE	NONE U	
	SET	L GE G LE E NE	Z	

Format

op rd=rb,rc

rd=op(rb,rc)
rd=opz(rcb)

31..24	23..18	17..12	11..6	5..0
A.MINOR	rd	rc	rb	op
8	6	6	6	6

rc ← rb ← rcb

Description

The contents of registers rc and rb are fetched and the specified operation is performed on these operands. The result is placed into register rd.

Definition

```
def AddressReversed(op,rd,rc,rb) as
    c ← RegRead(rc, 128)
    b ← RegRead(rb, 128)
    case op of
        ASET.E:
            a ← (b = c)^64
        ASET.NE:
            a ← (b ≠ c)^64
        ASET.AND.E:
            a ← ((b and c) = 0)^64
        ASET.AND.NE:
            a ← ((b and c) ≠ 0)^64
        ASET.L:
            a ← ((rc = rb) ? (b < 0) : (b < c))^64
        ASET.GE:
            a ← ((rc = rb) ? (b ≥ 0) : (b ≥ c))^64
        ASET.L.U:
```

```
            a ← ((rc = rb) ? (b > 0) : ((0 || b) < (0 || c))^64
        ASET.GE.U:
            a ← ((rc = rb) ? (b ≤ 0) : ((0 || b) ≥ (0 || c))^64
        ASUB:
            a ← b - c
        ASUB.O:
            t ← (b63 || b) - (c63 || c)
            if t64 ≠ t63 then
                raise FixedPointArithmetic
            endif
            a ← t63..0
        ASUB.U.O:
            t ← (0^1 || b) - (0^1 || c)
            if t64 ≠ 0 then
                raise FixedPointArithmetic
            endif
            a ← t63..0
    endcase
    RegWrite(rd, 64, a)
enddef
```

Exceptions

Fixed point arithmetic

Address Shift Left Immediate Add

These operations perform calculations with two general register values, placing the result in a general register.

Operation codes

ASHL.IADD	Address shift left immediate add

Format

ASHL.IADD rd=rc,rb,i

rc=op(ra,rb,i)

31 24	23 18	17 12	11 6	5 2	1 0
A.MINOR	rd	rc	rb	ASHL.IADD	sh
8	6	6	6	6	2

assert : $\leq i \leq 4$
sn ← . !

Description

The contents of register rb are shifted left by the immediate amount and added to the contents of register rc. The result is placed into register rd.

Definition

```
def AddressShiftLeftImmediateAdd(sh,rd,rc,rb) as
    c ← RegRead(rc, 64)
    b ← RegRead(rb, 64);
    a ← c + (b62-sh..0 || 0^(1+sh))
    RegWrite(rd, 64, a)
enddef
```

Exceptions

none

45

Address Shift Left Immediate Subtract

These operations perform calculations with two general register values, placing the result in a general register.

Operation codes

ASHL.I.SUB	Address shift left immediate subtract

Format

ASHL.I.SUB rd=rb,i,rc

rd=op(rb,i,rc)

31..24	23..18	17..12	11..6	5..2	1..0
A.MINOR	rd	rc	rb	ASHL.I.SUB	sh
8	6	6	6	6	2

assert $1 \le i \le 4$
sh ← i-1

Description

The contents of register rc is subtracted from the contents of register rb shifted left by the immediate amount. The result is placed into register rd.

Definition

def AddressShiftLeftImmediateSubtract(op,rd,rc,rb) as
    c ← RegRead(rc, 128)
    b ← RegRead(rb, 128)
    a ← $(b_{62-sh..0} \,||\, 0^{1+sh}) - c$
    RegWrite(rd, 64, a)
enddef

Exceptions

none

Address Shift Immediate

These operations perform calculations with one general register value and one immediate value, placing the result in a general register.

Operation codes

ASHL.I	Address shift left immediate
ASHL.I.O	Address shift left immediate signed check overflow
ASHL.I.U.O	Address shift left immediate unsigned check overflow
ASHR.I	Address signed shift right immediate
ASHR.I.U	Address shift right immediate unsigned

Redundancies

ASHL.I rd=rc,1	⇔	AADD rd=rc,rc
ASHL.I.O rd=rc,1	⇔	AADD.O rd=rc,rc
ASHL.I.U.O rd=rc,1	⇔	AADD.U.O rd=rc,rc
ASHL.I rd=rc,0	⇔	*ACOPY rd=rc*
ASHL.I.O rd=rc,0	⇔	*ACOPY rd=rc*
ASHL.I.U.O rd=rc,0	⇔	*ACOPY rd=rc*
ASHR.I rd=rc,0	⇔	*ACOPY rd=rc*
ASHR.I.U rd=rc,0	⇔	*ACOPY rd=rc*

Selection

class	operation	form	operand	check
shift	SHL	I		
			NONE U	O
	SHR	I	NONE U	

Format

op rd=rc,simm

rd=op(rc,simm)

31 … 24	23 … 18	17 … 12	11 … 6	5 … 0
A.MINOR	rd	rc	simm	op
8	6	6	6	6

Description

The contents of register rc is fetched, and a 6-bit immediate value is taken from the 6-bit simm field. The specified operation is performed on these operands. The result is placed into register rd.

Definition

def *AddressShiftImmediate(op,rd,rc,simm)* as
c ← *RegRead(rc, 64)*
case op of
ASHLI:
$a \leftarrow c_{63-simm..0} \mid\mid 0^{simm}$
ASHLI.O:
if $c_{63..63-simm} \neq c_{63}^{simm+1}$ then
raise FixedPointArithmetic
endif
$a \leftarrow c_{63-simm..0} \mid\mid 0^{simm}$
ASHLI.U.O:
if $c_{63..64-simm} \neq 0$ then
raise FixedPointArithmetic
endif
$a \leftarrow c_{63-simm..0} \mid\mid 0^{simm}$
ASHRI:
$a \leftarrow a_{63}^{simm} \mid\mid c_{63..simm}$
ASHRI.U:
$a \leftarrow 0^{simm} \mid\mid c_{63..simm}$
endcase
RegWrite(rd, 64, a)
enddef

Exceptions

Fixed-point arithmetic

Address Ternary

These operations perform calculations with three general register values, placing the result in a fourth general register.

Operation codes

AMUX	Address multiplex

Format

op ra=rd,rc,rb

ra=amux(rd,rc,rb)

31 24	23 18	17 12	11 6	5 0
op	rd	rc	rb	ra
8	6	6	6	6

Description

The contents of registers rd, rc, and rb are fetched. The specified operation is performed on these operands. The result is placed into register ra.

Definition

```
def AddressTernary(op,rd,rc,rb,ra) as
    d ← RegRead(rd, 64)
    c ← RegRead(rc, 64)
    b ← RegRead(rb, 64)
    endcase
    case op of
        AMUX:
            a ← (c and d) or (b and not d)
    endcase
    RegWrite(ra, 64, a)
enddef
```

Exceptions

none

Branch

This operation branches to a location specified by a register.

Operation codes

B	Branch

Format

B rd

31..24	23..18	17..12	11..6	5..0
B.MINOR	rd	0	0	B
8	6	6	6	6

Description

Execution branches to the address specified by the contents of register rd.

Access disallowed exception occurs if the contents of register rd is not aligned on a quadlet boundary.

Definition

```
def Branch(rd,rc,rb) as
    if (rc ≠ 0) or (rb ≠ 0) then
        raise ReservedInstruction
    endif
    d ← RegRead(rd, 64)
    if (d1..0) ≠ 0 then
        raise AccessDisallowedByVirtualAddress
    endif
    ProgramCounter ← d63..2 || 0^2
    raise TakenBranch
enddef
```

Exceptions

Reserved Instruction
Access disallowed by virtual address

Branch Back

This operation branches to a location specified by the previous contents of register 0, reduces the current privilege level, loads a value from memory, and restores register 0 to the value saved on a previous exception.

Operation codes

B.BACK	Branch back

Format

B.BACK

bback()

31..24	23..18	17..12	11..6	5..0
B.MINOR	0	0	0	B.BACK
8	6	6	6	6

Description

Processor context, including program counter and privilege level is restored from register 0, where it was saved at the last exception. Exception state, if set, is cleared, re-enabling normal exception handling. The contents of register 0 saved at the last exception is restored from memory. The privilege level is only lowered, so that this instruction need not be privileged.

If the previous exception was an AccessDetail exception, Continuation State set at the time of the exception affects the operation of the next instruction after this Branch Back, causing the previous AccessDetail exception to be inhibited. If software is performing this instruction to abort a sequence ending in an AccessDetail exception, it should abort by branching to an instruction that is not affected by Continuation State.

Definition

```
def BranchBack(rd,rc,rb) as
    c ← RegRead(rc, 128)
    if (rd ≠ 0) or (rc ≠ 0) or (rb ≠ 0) then
        raise ReservedInstruction
    endif
    a ← LoadMemory(ExceptionBase,ExceptionBase+Thread*128,128,L)
    if PrivilegeLevel > c1..0 then
        PrivilegeLevel ← c1..0
    endif
    ProgramCounter ← c63..2 || 0^2
    ExceptionState ← 0
    RegWrite(rd,128,a)
    raise TakenBranchContinue
enddef
```

Exceptions

Reserved Instruction
Access disallowed by virtual address
Access disallowed by tag
Access disallowed by global TB
Access disallowed by local TB
Access detail required by tag
Access detail required by local TB
Access detail required by global TB
Local TB miss
Global TB miss

Branch Barrier

This operation stops the current thread until all pending stores are completed, then branches to a location specified by a register.

Operation codes

B.BARRIER	Branch barrier

Format

B.BARRIER rd

bbarrier(rd)

31..24	23..18	17..12	11..6	5..0
B.MINOR	rd	0	0	B.BARRIER
8	6	6	6	6

Description

The instruction fetch unit is directed to cease execution until all pending stores are completed. Following the barrier, any previously pre-fetched instructions are discarded and execution branches to the address specified by the contents of register rd.

Access disallowed exception occurs if the contents of register rd is not aligned on a quadlet boundary.

Self-modifying, dynamically-generated, or loaded code may require use of this instruction between storing the code into memory and executing the code.

Definition

```
def BranchBarrier(rd,rc,rb) as
    if (rc ≠ 0) or (rb ≠ 0) then
        raise ReservedInstruction
    endif
    d ← RegRead(rd, 64)
    if (d1..0) ≠ 0 then
        raise AccessDisallowedByVirtualAddress
    endif
    ProgramCounter ← d63..2 || 0^2
    FetchBarrier()
    raise TakenBranch
enddef
```

Exceptions

Reserved Instruction

Branch Conditional

These operations compare two operands, and depending on the result of that comparison, conditionally branches to a nearby code location.

Operation codes

B.AND.E	Branch and equal zero
B.AND.NE	Branch and not equal zero
B.E	Branch equal
B.GE	Branch greater equal signed
B.L	Branch signed less
B.NE	Branch not equal
B.GE.U	Branch greater equal unsigned
B.L.U	Branch less unsigned

Equivalencies

B.E.Z	Branch equal zero
B.G.Z[6]	Branch greater zero signed
B.GE.Z[7]	Branch greater equal zero signed
B.L.Z[8]	Branch less zero signed
B.LE.Z[9]	Branch less equal zero signed
B.NE.Z	Branch not equal zero
B.LE	Branch less equal signed
B.G	Branch greater signed
B.LE.U	Branch less equal unsigned
B.G.U	Branch greater unsigned
B.NOP	Branch no operation

[6] B.G.Z is encoded as B.L.U with both instruction fields rd and rc equal.
[7] B.GE.Z is encoded as B.GE with both instruction fields rd and rc equal.
[8] B.L.Z is encoded as B.L with both instruction fields rd and rc equal.
[9] B.LE.Z is encoded as B.GE.U with both instruction fields rd and rc equal.

B.E.Z rc,target	←	B.AND.E rc,rc,target
B.G.Z rc,target	⇐	B.L.U rc,rc,target
B.GE.Z rc,target	⇐	B.GE rc,rc,target
B.L.Z rc,target	⇐	B.L rc,rc,target
B.LE.Z rc,target	⇐	B.GE.U rc,rc,target
B.NE.Z rc,target	←	B.AND.NE rc,rc,target
B.LE rc,rd,target	→	B.GE rd,rc,target
B.G rc,rd,target	→	B.L rd,rc,target
B.LE.U rc,rd,target	→	B.GE.U rd,rc,target
B.G.U rc,rd,target	→	B.L.U rd,rc,target
B.NOP	←	B.NE r0,r0,$

Redundancies

B.E rc,rc,target	⇔	B.I target
B.NE rc,rc,target	⇔	*B.NOP*

Selection

class	op	compare	type
arithmetic		L GE G *LE*	NONE U
vs. zero		*L GE G LE E NE*	*Z*
bitwise	none AND	E NE	

Format

op rd,rc,target

if (op(rd,rc)) goto target;

op	rd	rc	offset
31–24	23–18	17–12	11–0
8	6	6	12

Description

The contents of registers rd and rc are compared, as specified by the op field. If the result of the comparison is true, execution branches to the address specified by the offset field. Otherwise, execution continues at the next sequential instruction.

Definition

```
def BranchConditionally(op,rd,rc,offset) as
    d ← RegRead(rd, 128)
    c ← RegRead(rc, 128)
    case op of
        B.E:
```

```
            a ← d = c
        B.NE:
            a ← d ≠ c
        B.AND.E:
            a ← (d and c) = 0
        B.AND.NE:
            a ← (d and c) ≠ 0
        B.L:
            a ← (rd = rc) ? (c < 0): (d < c)
        B.GE:
            a ← (rd = rc) ? (c ≥ 0): (d ≥ c)
        B.L.U:
            a ← (rd = rc) ? (c > 0): ((0 || d) < (0 || c))
        B.GE.U:
            a ← (rd = rc) ? (c ≤ 0): ((0 || d) ≥ (0 || c))
    endcase
    if a then
        ProgramCounter ← ProgramCounter + (offset_11^50 || offset || 0^2)
        raise TakenBranch
    endif
enddef
```

Exceptions

none

Branch Conditional Floating-Point

These operations compare two floating-point operands, and depending on the result of that comparison, conditionally branches to a nearby code location.

Operation codes

B.E.F.16	Branch equal floating-point half
B.E.F.32	Branch equal floating-point single
B.E.F.64	Branch equal floating-point double
B.E.F.128	Branch equal floating-point quad
B.GE.F.16	Branch greater equal floating-point half
B.GE.F.32	Branch greater equal floating-point single
B.GE.F.64	Branch greater equal floating-point double
B.GE.F.128	Branch greater equal floating-point quad
B.L.F.16	Branch less floating-point half
B.L.F.32	Branch less floating-point single
B.L.F.64	Branch less floating-point double
B.L.F.128	Branch less floating-point quad
B.LG.F.16	Branch less greater floating-point half
B.LG.F.32	Branch less greater floating-point single
B.LG.F.64	Branch less greater floating-point double
B.LG.F.128	Branch less greater floating-point quad

Equivalencies

B.LE.F.16	Branch less equal floating-point half
B.LE.F.32	Branch less equal floating-point single
B.LE.F.64	Branch less equal floating-point double
B.LE.F.128	Branch less equal floating-point quad
B.G.F.16	Branch greater floating-point half
B.G.F.32	Branch greater floating-point single
B.G.F.64	Branch greater floating-point double
B.G.F.128	Branch greater floating-point quad

B.LE.F.size rc,rd,target	→	B.GE.F.size rd,rc,target
B.G.F.size rc,rd,target	→	B.L.F.size rd,rc,target

Selection

number format	type	compare	size
floating-point	F	E LG L GE *G* *LE*	16 32 64 128

51

Format

op rd,rc,target

if (op(rd,rc)) goto target;

31 ... 24	23 ... 18	17 ... 12	11 ... 0
op	rd	rc	offset
8	6	6	12

Description

The contents of registers rc and rd are compared, as specified by the op field. If the result of the comparison is true, execution branches to the address specified by the offset field. Otherwise, execution continues at the next sequential instruction.

Definition

```
def BranchConditionalFloatingPoint(op,rd,rc,offset) as
    case op of
        B.E.F.16, B.LG.F.16, B.L.F.16, B.GE.F.16:
            size ← 16
        B.E.F.32, B.LG.F.32, B.L.F.32, B.GE.F.32:
            size ← 32
        B.E.F.64, B.LG.F.64, B.L.F.64, B.GE.F.64:
            size ← 64
        B.E.F.128, B.LG.F.128, B.L.F.128, B.GE.F.128:
            size ← 128
    endcase
    d ← F(size,RegRead(rd, 128))
    c ← F(size,RegRead(rc, 128))
    v ← fcom(d, c)
    case op of
        BEF16, BEF32, BEF64, BEF128:
            a ← (v = E)
        BLGF16, BLGF32, BLGF64, BLGF128:
            a ← (v = L) or (v = G)
        BLF16, BLF32, BLF64, BLF128:
            a ← (v = L)
        BGEF16, BGEF32, BGEF64, BGEF128:
            a ← (v = G) or (v = E)
    endcase
    if a then
        ProgramCounter ← ProgramCounter + (offset[11]^50 || offset || 0^2)
        raise TakenBranch
    endif
enddef
```

Exceptions

none

Branch Conditional Visibility Floating-Point

These operations compare two group-floating-point operands, and depending on the result of that comparison, conditionally branches to a nearby code location.

Operation codes

B.I.F.32	Branch invisible floating-point single
B.NI.F.32	Branch not invisible floating-point single
B.NV.F.32	Branch not visible floating-point single
B.V.F.32	Branch visible floating-point single

Selection

number format	type	compare	size
floating-point	F	I NI NV V	32

Format

op rc,rd,target

if (op(rc,rd)) goto target;

31 … 24	23 … 18	17 … 12	11 … 0
op	rd	rc	offset
8	6	6	12

Description

The contents of registers rc and rd are compared, as specified by the op field. If the result of the comparison is true, execution branches to the address specified by the offset field. Otherwise execution continues at the next sequential instruction.

Each operand is assumed to represent a vertex of the form: [w x y z] packed into a single register. The comparisons check for visibility of a line connecting the vertices against a standard viewing volume, defined by the planes: x=w,x=-w,y=w,y=-w,z=0,z=1. A line is visible (V) if the vertices are both within the volume. A line is not visible (NV) is either vertex is outside the volume - in such a case, the line may be partially visible. A line is invisible (I) if the vertices are both outside any face of the volume. A line is not invisible (NI) if the vertices are not both outside any face of the volume.

Definition

def n(a) as (a.t=QNAN) or (a.t=SNAN) enddef

def less(a,b) as fcom(a,b)=L enddef

def trxya,b,c,d) as (fcom(fabs(a),b)=G) and (fcom(fabs(c),d)=G) and (a.s=c.s) enddef

```
def BranchConditionalVisibilityFloatingPoint(op,rd,rc,offset) as
    d ← RegRead(rd, 128)
    c ← RegRead(rc, 128)
    dx ← F(32,d31..0)
    cx ← F(32,c31..0)
    dy ← F(32,d63..32)
    cy ← F(32,c63..32)
    dz ← F(32,d95..64)
    cz ← F(32,c95..64)
    dw ← F(32,d127..96)
    cw ← F(32,c127..96)
    f1 ← F(32,0x7f000000) // floating-point 1.0
    if (n(dx) or n(dy) or n(dz) or n(dw) or n(cx) or n(cy) or n(cz) or n(cw)) then
        a ← false
    else
        dv ← less(fabs(dx),dz) and less(fabs(dy),dz) and less(dz,f1) and (dz.s=0)
        cv ← less(fabs(cx),cz) and less(fabs(cy),cz) and less(cz,f1) and (cz.s=0)
        trz ← (less(f1,dz) and less(f1,cz)) or ((dz.s=1 and cz.s=1))
        tr ← trxy(dx,dz,cx,cz) or trxy(dy,dz,cy,cz) or trz
        case op of
            B.I.F.32:
                a ← tr
            B.NI.F.32:
                a ← not tr
            B.NV.F.32:
                a ← not (dv and cv)
            B.V.F.32:
                a ← dv and cv
        endcase
    endif
    if a then
        ProgramCounter ← ProgramCounter + (offset^50 || offset || 0^2)
        raise TakenBranch
    endif
enddef
```

Exceptions

none

Branch Down

This operation branches to a location specified by a register, reducing the current privilege level

Operation codes

B.DOWN	Branch down

Format

B.DOWN rd

bdown(rd)

31 24	23 18	17 12	11 6	5 0
B.MINOR	rd	0	0	B.DOWN
8	6	6	6	6

Description

Execution branches to the address specified by the contents of register rd. The current privilege level is reduced to the level specified by the low order two bits of the contents of register rd.

Definition

```
def BranchDown(rd,rc,rb) as
    if (rc ≠ 0) or (rb ≠ 0) then
        raise ReservedInstruction
    endif
    d ← RegRead(rd, 64)
    if PrivilegeLevel > d1..0 then
        PrivilegeLevel ← d1..0
    endif
    ProgramCounter ← d63..2 || 0^2
    raise TakenBranch
enddef
```

Exceptions

Reserved Instruction

Branch Gateway

This operation provides a secure means to call a procedure, including those at a higher privilege level.

Operation codes

B.GATE	Branch gateway

Equivalencies

B.GATE	←	B.GATE 0

Format

B.GATE rb

bgate(rb)

31..24	23..18	17..12	11..6	5..0
B.MINOR	0	1	rb	B.GATE
8	6	6	6	6

Description

The contents of register rb is a branch address in the high-order 62 bits and a new privilege level in the low-order 2 bits. A branch and link occurs to the branch address, and the privilege level is raised to the new privilege level. The high-order 62 bits of the successor to the current program counter is catenated with the 2-bit current execution privilege and placed in register 0.

If the new privilege level is greater than the current privilege level, an octlet of memory data is fetched from the address specified by register 1, using the little-endian byte order and a gateway access type. A GatewayDisallowed exception occurs if the original contents of register 0 do not equal the memory data.

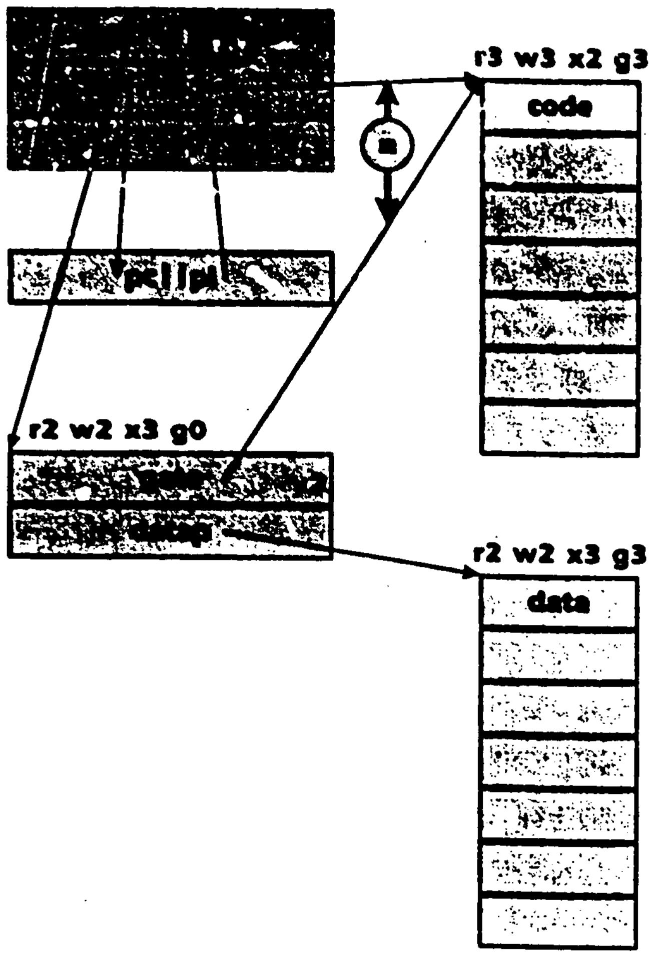
If the new privilege level is the same as the current privilege level, no checking of register 1 is performed.

An AccessDisallowed exception occurs if the new privilege level is greater than the privilege level required to write the memory data, or if the old privilege level is lower than the privilege required to access the memory data as a gateway, or if the access is not aligned on an 8-byte boundary.

A ReservedInstruction exception occurs if the rc field is not one or the rd field is not zero.

In the example below, a gateway from level 0 to level 2 is illustrated. The gateway pointer, located by the contents of register rc (1), is fetched from memory and compared against the

contents of register rb (0). The instruction may only complete if these values are equal. Concurrently, the contents of register rb (0) is placed in the program counter and privilege level, and the address of the next sequential address and privilege level is placed into register rd (0). Code at the target of the gateway locates the data pointer at an offset from the gateway pointer (register 1), and fetches it into register 1, making a data region available. A stack pointer may be saved and fetched using the data region, another region located from the data region, or a data region located as an offset from the original gateway pointer.



Branch gateway

For additional information on the branch-gateway instruction, see the System and Privileged Library Calls section on page 44.

This instruction gives the target procedure the assurances that register 0 contains a valid return address and privilege level, that register 1 points to the gateway location, and that the gateway location is octlet aligned. Register 1 can then be used to

securely reach values in memory. If no sharing of literal pools is desired, register 1 may be used as a literal pool pointer directly. If sharing of literal pools is desired, register 1 may be used with an appropriate offset to load a new literal pool pointer; for example, with a one cache line offset from the register 1. Note that because the virtual memory system operates with cache line granularity, that several gateway locations must be created together.

Software must ensure that an attempt to use any octlet within the region designated by virtual memory as gateway either functions properly or causes a legitimate exception. For example, if the adjacent octlets contain pointers to literal pool locations, software should ensure that these literal pools are not executable, or that by virtue of being aligned addresses, cannot raise the execution privilege level. If register 1 is used directly as a literal pool location, software must ensure that the literal pool locations that are accessible as a gateway do not lead to a security violation.

Register 0 contains a valid return address and privilege level, the value is suitable for use directly in the Branch-down (B.DOWN) instruction to return to the gateway callee.

Definition

```
def BranchGateway(rd,rc,rb) as
    c ← RegRead(rc, 64)
    b ← RegRead(rb, 64)
    if (rd = 0) or (rc = 1) then
        raise ReservedInstruction
    endif
    if c2..0 ≠ 0 then
        raise AccessDisallowedByVirtualAddress
    endif
    d ← ProgramCounter63..2+1 || PrivilegeLevel
    if PrivilegeLevel < b1..0 then
        m ← LoadMemoryG(c,c,64,L)
        if b ≠ m then
            raise GatewayDisallowed
        endif
        PrivilegeLevel ← b1..0
    endif
    ProgramCounter ← b63..2 || 0^2
    RegWrite(rd, 64, d)
    raise TakenBranch
enddef
```

Exceptions

Reserved Instruction
Gateway disallowed
Access disallowed by virtual address
Access disallowed by tag
Access disallowed by global TB
Access disallowed by local TB
Access detail required by tag

Access detail required by local TB
Access detail required by global TB
Local TB miss
Global TB miss

Branch Halt

This operation stops the current thread until an exception occurs.

Operation codes

B.HALT	Branch halt

Format

B.HALT

bhalt()

31 24	23 18	17 12	11 6	5 0
B.MINOR	0	0	0	B.HALT
8	6	6	6	6

Description

This instruction directs the instruction fetch unit to cease execution until an exception occurs.

Definition

```
def BranchHalt(rd,rc,rb) as
    if (rd ≠ 0) or (rc ≠ 0) or (rb ≠ 0) then
        raise ReservedInstruction
    endif
    FetchHalt()
enddef
```

Exceptions

Reserved Instruction

Branch Hint

This operation indicates a future branch location specified by a register

Operation codes

B.HINT	Branch Hint

Format

B.HINT badd,count,rd

bhint(badd,count,rd)

31 24	23 18	17 12	11 6	5 0
B.MINOR	rd	count	simm	B.HINT
8	6	6	6	6

simm ← badd-pc-4

Description

This instruction directs the instruction fetch unit of the processor that a branch is likely to occur count times at simm instructions following the current successor instruction to the address specified by the contents of register rd.

After branching count times, the instruction fetch unit should presume that the branch at simm instructions following the current successor instruction is not likely to occur. If count is zero, this hint directs the instruction fetch unit that the branch is likely to occur more than 63 times.

Access disallowed exception occurs if the contents of register rd is not aligned on a quadlet boundary

Definition

```
def BranchHint(rd,count,simm) as
    d ← RegRead(rd, 64)
    if (d1..0) ≠ 0 then
        raise AccessDisallowedByVirtualAddress
    endif
    FetchHint(ProgramCounter + 4 + (0 || simm || 0^2), d63..2 || 0^2, count)
enddef
```

Exceptions

Access disallowed by virtual address

Branch Hint Immediate

This operation indicates a future branch location specified as an offset from the program counter.

Operation codes

B.HINT.I	Branch Hint Immediate

Format

B.HINT.I badd,count,target

bhinti(badd,count,target)

31 24	23 18	17 12	11 0
B.HINT.I	simm	count	offset
8	6	6	12

simm ← badd-pc-4

Description

This instruction directs the instruction fetch unit of the processor that a branch is likely to occur count times at simm instructions following the current successor instruction to the address specified by the offset field.

After branching count times, the instruction fetch unit should presume that the branch at simm instructions following the current successor instruction is not likely to occur. If count is zero, this hint directs the instruction fetch unit that the branch is likely to occur more than 63 times.

Definition

```
def BranchHintImmediate(simm,count,offset) as
    BranchHint(ProgramCounter + 4 + (0 || simm || 0^2), count,
            ProgramCounter + (offset{}} || offset || 0^2))
enddef
```

Exceptions

none

Branch Immediate

This operation branches to a location that is specified as an offset from the program counter.

Operation codes

B.I	Branch immediate

Redundancies

B.I target	⇔	B.E rc,rc,target

Format

B.I target

bi(target)

31 24	23 0
B.I	offset
8	24

Description

Execution branches to the address specified by the offset field.

Definition

```
def BranchImmediate(offset) as
    ProgramCounter ← ProgramCounter + (offset_23^8 || offset || 0^2)
    raise TakenBranch
enddef
```

Exceptions

none

57

Branch Immediate Link

This operation branches to a location that is specified as an offset from the program counter, saving the value of the program counter into register 0.

Operation codes

B.LINK.I	Branch immediate link

Format

B.LINK.I target

blinki(target)

31 24	23 0
B.LINK.I	offset
8	24

Description

The address of the instruction following this one is placed into register 0. Execution branches to the address specified by the offset field.

Definition

```
def BranchImmediateLink(offset) as
    RegWrite(0, 64, ProgramCounter + 4)
    ProgramCounter ← ProgramCounter + (offset_23^38 || offset || 0^2)
    raise TakenBranch
enddef
```

Exceptions

none

Branch Link

This operation branches to a location specified by a register, saving the value of the program counter into a register.

Operation codes

B.LINK	Branch link

Equivalencies

B.LINK	←	B.LINK 0=0
B.LINK rc	←	B.LINK 0=rc

Format

B.LINK rd=rc

31..24	23..18	17..12	11..6	5..0
B.MINOR	rd	rc	0	B.LINK
8	6	6	6	6

rb ← 0

Description

The address of the instruction following this one is placed into register rd. Execution branches to the address specified by the contents of register rc.

Access disallowed exception occurs if the contents of register rc is not aligned on a quadlet boundary.

Reserved instruction exception occurs if rb is not zero.

Definition

```
def BranchLink(rd,rc,rb) as
    if rb ≠ 0 then
        raise ReservedInstruction
    endif
    c ← RegRead(rc, 64)
    if (c and 3) ≠ 0 then
        raise AccessDisallowedByVirtualAddress
    endif
    RegWrite(rd, 64, ProgramCounter + 4)
    ProgramCounter ← c63..2 || 0^2
    raise TakenBranch
enddef
```

Exceptions

Reserved Instruction
Access disallowed by virtual address

Load

These operations compute a virtual address from the contents of two registers, load data from memory, sign- or zero-extending the data to fill the destination register.

Operation codes

L8[10]	Load signed byte
L.16.B	Load signed doublet big-endian
L.16AB	Load signed doublet aligned big-endian
L.16.L	Load signed doublet little-endian
L.16AL	Load signed doublet aligned little-endian
L32.B	Load signed quadlet big-endian
L32AB	Load signed quadlet aligned big-endian
L32.L	Load signed quadlet little-endian
L32AL	Load signed quadlet aligned little-endian
L64.B	Load signed octlet big-endian
L64AB	Load signed octlet aligned big-endian
L.64.L	Load signed octlet little-endian
L64AL	Load signed octlet aligned little-endian
L128.B[11]	Load hexlet big-endian
L128AB[12]	Load hexlet aligned big-endian
L128.L[13]	Load hexlet little-endian
L128AL[14]	Load hexlet aligned little-endian
LU.8[15]	Load unsigned byte
LU.16.B	Load unsigned doublet big-endian
LU.16AB	Load unsigned doublet aligned big-endian
LU.16L	Load unsigned doublet little-endian
LU.16AL	Load unsigned doublet aligned little-endian
LU.32.B	Load unsigned quadlet big-endian
LU32AB	Load unsigned quadlet aligned big-endian
LU.32.L	Load unsigned quadlet little-endian
LU32AL	Load unsigned quadlet aligned little-endian
LU.64.B	Load unsigned octlet big-endian
LU.64AB	Load unsigned octlet aligned big-endian
LU.64.L	Load unsigned octlet little-endian
LU.64AL	Load unsigned octlet aligned little-endian

[10] L8 need not distinguish between little endian and big endian ordering, nor between aligned and unaligned, as only a single byte is loaded.

[11] L128.B need not distinguish between signed and unsigned, as the hexlet fills the destination register.

[12] L128.AB need not distinguish between signed and unsigned, as the hexlet fills the destination register.

[13] L128.L need not distinguish between signed and unsigned, as the hexlet fills the destination register.

[14] L128.AL need not distinguish between signed and unsigned, as the hexlet fills the destination register.

[15] LU8 need not distinguish between little endian and big-endian ordering, nor between aligned and unaligned, as only a single byte is loaded.

Selection

number format	type	size	alignment	ordering
signed byte		8		
unsigned byte	U	8		
signed integer		16 32 64		L B
signed integer aligned		16 32 64	A	L B
unsigned integer	U	16 32 64		L B
unsigned integer aligned	U	16 32 64	A	L B
register		128		L B
register aligned		128	A	L B

Format

op rd=rc,rb

rd=op(rc,rb)

31 24	23 18	17 12	11 6	5 0
L.MINOR	rd	rc	rb	op
8	6	6	6	6

Description

An operand size, expressed in bytes, is specified by the instruction. A virtual address is computed from the sum of the contents of register rc and the contents of register rb multiplied by operand size. The contents of memory using the specified byte order are read, treated as the size specified, zero-extended or sign-extended as specified, and placed into register rd.

If alignment is specified, the computed virtual address must be aligned, that is, it must be an exact multiple of the size expressed in bytes. If the address is not aligned an "access disallowed by virtual address" exception occurs.

Definition

```
def Load(op,rd,rc,rb) as
    case op of
        L16L, L32L, L8, L16AL, L32AL, L16B, L32B, L16AB, L32AB,
        L64L, L64AL, L64B, L64AB:
            signed ← true
        LU16L, LU32L, LU8, LU16AL, LU32AL, LU16B, LU32B, LU16AB, LU32AB,
        LU64L, LU64AL, LU64B, LU64AB:
            signed ← false
        L128L, L128AL, L128B, L128AB:
            signed ← undefined
    endcase
    case op of
        L8, LU8:
            size ← 8
        L16L, LU16L, L16AL, LU16AL, L16B, LU16B, L16AB, LU16AB:
```

```
            size ← 16
        L32L, LU32L, L32AL, LU32AL, L32B, LU32B, L32AB, LU32AB:
            size ← 32
        L64L, LU64L, L64AL, LU64AL, L64B, LU64B, L64AB, LU64AB:
            size ← 64
        L128L, L128AL, L128B, L128AB:
            size ← 128
    endcase
    lsize ← log(size)
    case op of
        L16L, LU16L, L32L, LU32L, L64L, LU64L, L128L,
        L16AL, LU16AL, L32AL, LU32AL, L64AL, LU64AL, L128AL:
            order ← L
        L16B, LU16B, L32B, LU32B, L64B, LU64B, L128B,
        L16AB, LU16AB, L32AB, LU32AB, L64AB, LU64AB, L128AB:
            order ← B
        L8, LU8:
            order ← undefined
    endcase
    c ← RegRead(rc, 64)
    b ← RegRead(rb, 64)
    VirtAddr ← c + (b63-lsize..0 || 0^(lsize-3))
    case op of
        L16AL, LU16AL, L32AL, LU32AL, L64AL, LU64AL, L128AL,
        L16AB, LU16AB, L32AB, LU32AB, L64AB, LU64AB, L128AB:
            if (c_(lsize-4..0) ≠ 0 then
                raise AccessDisallowedByVirtualAddress
            endif
        L16L, LU16L, L32L, LU32L, L64L, LU64L, L128L,
        L16B, LU16B, L32B, LU32B, L64B, LU64B, L128B:
        L8, LU8:
    endcase
    m ← LoadMemory(c,VirtAddr,size,order)
    a ← (m_(size-1) and signed)^(128-size) || m
    RegWrite(rd, 128, a)
enddef
```

Exceptions

Access disallowed by virtual address
Access disallowed by tag
Access disallowed by global TB
Access disallowed by local TB
Access detail required by tag
Access detail required by local TB
Access detail required by global TB
Local TB miss
Global TB miss

Load Immediate

These operations compute a virtual address from the contents of a register and a sign-extended immediate value, load data from memory, sign- or zero-extending the data to fill the destination register.

Operation codes

L.I.8[16]	Load immediate signed byte
L.I.16AB	Load immediate signed doublet aligned big-endian
L.I.16.B	Load immediate signed doublet big-endian
L.I.16AL	Load immediate signed doublet aligned little-endian
L.I.16.L	Load immediate signed doublet little-endian
L.I.32AB	Load immediate signed quadlet aligned big-endian
L.I.32.B	Load immediate signed quadlet big-endian
L.I.32AL	Load immediate signed quadlet aligned little-endian
L.I.32.L	Load immediate signed quadlet little-endian
L.I.64AB	Load immediate signed octlet aligned big-endian
L.I.64.B	Load immediate signed octlet big-endian
L.I.64AL	Load immediate signed octlet aligned little-endian
L.I.64.L	Load immediate signed octlet little-endian
L.I.128AB[17]	Load immediate hexlet aligned big-endian
L.I.128.B[18]	Load immediate hexlet big-endian
L.I.128AL[19]	Load immediate hexlet aligned little-endian
L.I.128.L[20]	Load immediate hexlet little-endian
L.I.U.8[21]	Load immediate unsigned byte
L.I.U.16AB	Load immediate unsigned doublet aligned big-endian
L.I.U.16.B	Load immediate unsigned doublet big-endian
L.I.U.16AL	Load immediate unsigned doublet aligned little-endian
L.I.U.16.L	Load immediate unsigned doublet little-endian
L.I.U.32AB	Load immediate unsigned quadlet aligned big-endian
L.I.U.32.B	Load immediate unsigned quadlet big-endian
L.I.U.32AL	Load immediate unsigned quadlet aligned little-endian
L.I.U.32.L	Load immediate unsigned quadlet little-endian
L.I.U.64AB	Load immediate unsigned octlet aligned big-endian
L.I.U.64.B	Load immediate unsigned octlet big-endian
L.I.U.64AL	Load immediate unsigned octlet aligned little-endian
L.I.U.64.L	Load immediate unsigned octlet little-endian

[16] L.I.8 need not distinguish between little-endian and big-endian ordering, nor between aligned and unaligned, as only a single byte is loaded.

[17] L.I.128.AB need not distinguish between signed and unsigned, as the hexlet fills the destination register.

[18] L.I.128.B need not distinguish between signed and unsigned, as the hexlet fills the destination register.

[19] L.I.128.AL need not distinguish between signed and unsigned, as the hexlet fills the destination register.

[20] L.I.128.L need not distinguish between signed and unsigned, as the hexlet fills the destination register.

[21] L.I.U8 need not distinguish between little-endian and big-endian ordering, nor between aligned and unaligned, as only a single byte is loaded.

Selection

number format	type	size	alignment	ordering
signed byte		8		
unsigned byte	U	8		
signed integer		16 32 64		L B
signed integer aligned		16 32 64	A	L B
unsigned integer	U	16 32 64		L B
unsigned integer aligned	U	16 32 64	A	L B
register		128		L B
register aligned		128	A	L B

Format

op rd=rc,offset

rd=op(rc,offset)

31 24	23 18	17 12	11 0
op	rd	rc	offset
8	6	6	12

Description

An operand size, expressed in bytes, is specified by the instruction. A virtual address is computed from the sum of the contents of register rc and the sign-extended value of the offset field, multiplied by the operand size. The contents of memory using the specified byte order are read, treated as the size specified, zero-extended or sign-extended as specified, and placed into register rd.

If alignment is specified, the computed virtual address must be aligned, that is, it must be an exact multiple of the size expressed in bytes. If the address is not aligned an "access disallowed by virtual address" exception occurs.

Definition

```
def LoadImmediate(op,rd,rc,offset) as
    case op of
        LI16L, LI32L, LI8, LI16AL, LI32AL, LI16B, LI32B, LI16AB, LI32AB:
        LI64L, LI64AL, LI64B, LI64AB:
            signed ← true
        LIU16L, LIU32L, LIU8, LIU16AL, LIU32AL,
        LIU16B, LIU32B, LIU16AB, LIU32AB:
        LIU64L, LIU64AL, LIU64B, LIU64AB:
            signed ← false
        LI128L, LI128AL, LI128B, LI128AB:
            signed ← undefined
    endcase
    case op of
        LI8, LIU8:
            size ← 8
        LI16L, LIU16L, LI16AL, LIU16AL, LI16B, LIU16B, LI16AB, LIU16AB:
            size ← 16
```

61

```
        LI32L, LIU32L, LI32AL, LIU32AL, LI32B, LIU32B, LI32AB, LIU32AB:
            size ← 32
        LI64L, LIU64L, LI64AL, LIU64AL, LI64B, LIU64B, LI64AB, LIU64AB:
            size ← 64
        LI128L, LI128AL, LI128B, LI128AB:
            size ← 128
    endcase
    lsize ← log(size)
    case op of
        LI16L, LIU16L, LI32L, LIU32L, LI64L, LIU64L, LI128L,
        LI16AL, LIU16AL, LI32AL, LIU32AL, LI64AL, LIU64AL, LI128AL:
            order ← L
        LI16B, LIU16B, LI32B, LIU32B, LI64B, LIU64B, LI128B,
        LI16AB, LIU16AB, LI32AB, LIU32AB, LI64AB, LIU64AB, LI128AB:
            order ← B
        LI8, LIU8:
            order ← undefined
    endcase
    c ← RegRead(rc, 64)
    VirtAddr ← c + (offset[55-lsize] || offset || 0[lsize-3])
    case op of
        LI16AL, LIU16AL, LI32AL, LIU32AL, LI64AL, LIU64AL, LI128AL,
        LI16AB, LIU16AB, LI32AB, LIU32AB, LI64AB, LIU64AB, LI128AB:
            if c[lsize-4..0] ≠ 0 then
                raise AccessDisallowedByVirtualAddress
            endif
        LI16L, LIU16L, LI32L, LIU32L, LI64L, LIU64L, LI128L,
        LI16B, LIU16B, LI32B, LIU32B, LI64B, LIU64B, LI128B:
        LI8, LIU8:
    endcase
    m ← LoadMemory(c,VirtAddr,size,order)
    a ← (m[size-1] and signed)^(128-size) || m
    RegWrite(rd, 128, a)
enddef
```

Exceptions

Access disallowed by virtual address
Access disallowed by tag
Access disallowed by global TB
Access disallowed by local TB
Access detail required by tag
Access detail required by local TB
Access detail required by global TB
Local TB miss
Global TB miss